

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

LÊ HOÀNG OANH
NGUYỄN BÌNH KHẢI

BÁO CÁO ĐÒ ÁN CHUYÊN NGÀNH

ĐIỀU PHỐI CONTAINER VÀ QUẢN LÝ
TÀI NGUYÊN CHO CÁC TÁC VỤ HỌC
MÁY VỚI KUBERNETES

CONTAINER ORCHESTRATION AND
RESOURCE MANAGEMENT FOR
MACHINE LEARNING TASKS WITH
KUBERNETES

TP. HỒ CHÍ MINH, 2025

**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG**

LÊ HOÀNG OANH - 21521253

NGUYỄN BÌNH KHẢI - 21522184

BÁO CÁO ĐỒ ÁN CHUYÊN NGÀNH

**ĐIỀU PHỐI CONTAINER VÀ QUẢN LÝ
TÀI NGUYÊN CHO CÁC TÁC VỤ HỌC
MÁY VỚI KUBERNETES**

**CONTAINER ORCHESTRATION AND
RESOURCE MANAGEMENT FOR
MACHINE LEARNING TASKS WITH
KUBERNETES**

**GIẢNG VIÊN HƯỚNG DẪN
THS. ĐỖ HOÀNG HIỀN**

TP. HỒ CHÍ MINH, 2025

LỜI CẢM ƠN

Lời đầu tiên, chúng tôi xin chân thành cảm ơn các thầy cô, đặc biệt là ThS. Đỗ Hoàng Hiển, giảng viên hướng dẫn, người đã tận tâm, nhiệt tình chỉ bảo và hỗ trợ chúng tôi trong suốt quá trình thực hiện đồ án chuyên ngành này. Sự hướng dẫn tận tình của thầy đã giúp chúng tôi hiểu rõ hơn về các kiến thức chuyên môn cũng như cách áp dụng chúng vào thực tiễn, giúp chúng tôi hoàn thành đồ án một cách suôn sẻ.

Chúng tôi cũng xin cảm ơn Trường Đại học Công nghệ Thông tin, Đại học Quốc gia Thành phố Hồ Chí Minh, Khoa Mạng Máy tính và Truyền thông đã tạo điều kiện về cơ sở vật chất, tài liệu học tập và môi trường học thuật thuận lợi, giúp chúng tôi có cơ hội tiếp cận những công nghệ tiên tiến và phát triển tư duy nghiên cứu khoa học.

Cuối cùng, chúng tôi xin cảm ơn gia đình và bạn bè đã luôn động viên, hỗ trợ chúng tôi trong suốt quá trình học tập và thực hiện đồ án. Sự ủng hộ và động viên của các bạn là nguồn động lực lớn lao giúp chúng tôi vượt qua mọi khó khăn.

MỤC LỤC

Chương 1. Giới thiệu đề tài	3
1.1. Giới thiệu vấn đề	3
1.1.1. Hiện trạng	3
1.1.2. Thách thức	3
1.2. Mục tiêu của đồ án	4
1.3. Phương pháp nghiên cứu	4
Chương 2. Cơ sở lý thuyết	5
2.1. Container	5
2.1.1. Khái niệm	5
2.1.2. Đặc điểm của Container	5
2.2. Docker	5
2.2.1. Khái niệm	5
2.2.2. Kiến trúc của Docker	6
2.2.3. Quy trình thực thi	6
2.3. Kubernetes	7
2.3.1. Kiến trúc chính	7
2.3.2. Các thành phần của Control Plane.....	8
2.3.2.1. Kube-API Server:	8
2.3.2.2. Etcd:	8
2.3.2.3. Kube-Scheduler:	9
2.3.2.4. Kube-Controller-Manager	9
2.3.2.5. Cloud-Controller-Manager (Tùy chọn)	9
2.3.3. Các thành phần của Worker Node.....	9

2.3.3.1. Kubelet.....	10
2.3.3.2. Kube-Proxy	10
2.3.3.3. Container Runtime	10
2.3.4. Add-ons trong Kubernetes.....	10
2.3.4.1. DNS.....	10
2.3.4.2. Dashboard	10
2.3.4.3. Ingress Controller	11
2.3.4.4. Metrics Server.....	11
2.3.4.5. Network Plugins	11
2.3.4.6. Service và Ingress	11
2.3.5. Workload trong Kubernetes	12
2.3.5.1. Pod	12
2.3.5.2. Deployment.....	13
2.3.5.3. StatefulSet	13
2.3.5.4. DaemonSet.....	14
2.3.5.5. Job	14
2.3.5.6. CronJob	14
2.3.6. Storage trong Kubernetes	15
2.3.6.1. Tổng quan	15
2.3.6.2. Các khái niệm quan trọng	15
2.3.7. Cách hoạt động của Kubernetes:	17
2.4. High Availability trong Kubernetes	17
2.4.1. Các Thành Phần Chính trong HA.....	18
2.4.1.1. Master Node Redundancy:.....	18

2.4.1.2. Worker Node Redundancy:	18
2.4.1.3. Etcd Clustering:	19
2.4.1.4. Topologies:	19
2.4.2. Các Thành Phần Cần Thiết để Đảm Bảo HA trong Kubernetes .	20
2.4.3. Kỹ Thuật và Công Cụ Hỗ Trợ HA trong Kubernetes	21
2.4.3.1. Keepalived và Kube-VIP	21
2.4.3.2. Luồng Kết Nối với VIP và HA Control Plane	21
2.5. RKE2	22
2.5.1.1. Kiến trúc.....	23
2.5.1.2. Các công nghệ được tích hợp sẵn	23
2.6. So sánh các công cụ.....	24
2.6.1. So sánh Docker và Kubernetes	24
2.6.2. So sánh các loại cluster.....	26
Chương 3. Phân tích thiết kế hệ thống	29
3.1. Vấn đề.....	29
3.2. Phương pháp	29
3.2.1. Tìm hiểu các tính năng của Docker và Kubernetes.....	29
3.2.2. Cài đặt Docker và Kubernetes	30
3.2.3. Tự động hóa việc tạo Cluster và khám phá các tính năng.....	30
3.2.4. Cài đặt và khởi tạo RKE, RKE2 Cluster	30
3.2.5. Tìm hiểu về Rancher và các công cụ tích hợp.....	31
3.2.6. Cài đặt High Availability (HA) cho Master Node	31
3.2.7. Cài đặt và cấu hình LDAP Server	32
3.2.8. Cài đặt các công cụ DevOps.....	32

3.2.9.	Tổng kết phương pháp.....	33
3.3.	Kiến trúc	33
3.3.1.	Kiến trúc triển khai RKE2 Cluster	33
3.3.1.1.	Ansible Control Machine.....	33
3.3.1.2.	Virtual Machine (VM)	34
3.3.2.	Kiến trúc triển khai các ứng dụng quản lí cluster.....	34
3.3.2.1.	Kubernetes Cluster.....	34
3.3.2.2.	Quản lý Tài Nguyên.....	35
3.3.2.3.	Quản lý Docker Images (Docker Registry và UI)	35
3.3.2.4.	Giám Sát và Logging (Kube-Prometheus Stack).	35
3.3.3.	Kiến trúc triển khai các ứng dụng học máy trong cluster.....	36
3.3.3.1.	Quản lý và Triển khai Mô Hình Học Máy.....	36
3.3.3.2.	Quản lý Môi Trường Làm Việc (JupyterHub).....	36
3.4.	Luồng triển khai RKE2 Cluster	37
3.4.1.	Viết Ansible Playbook.....	37
3.4.2.	Push Ansible Playbook lên GitHub	38
3.4.3.	Clone Ansible Playbook về Ansible Control Machine	38
3.4.4.	Chạy Ansible Playbook trên Ansible Control Machine	38
3.4.5.	Deploy RKE2 lên các Master và Worker Node	38
3.5.	Luồng hoạt động của các ứng dụng trong Cluster.....	39
3.5.1.	RKE2 HA	39
3.5.1.1.	Keepalived và VRRP VIP:	39
3.5.1.2.	Luồng giao tiếp:	39
3.5.2.	Rancher	40

3.5.2.1. Xác thực người dùng.....	40
3.5.2.2. Cluster Controller và Cluster Agent	40
3.5.2.3. Node Agent	41
3.5.2.4. Kết nối trực tiếp (ACE).....	41
3.5.3. Longhorn	41
3.5.4. Docker Registry	43
3.5.4.1. Lưu trữ và Phân phối	43
3.5.4.2. Quá trình Pull và Push	43
3.5.5. Metallb.....	44
3.5.5.1. Cáp phát địa chỉ IP (IP Pool)	44
3.5.5.2. Giao tiếp dịch vụ.....	44
3.5.5.3. Chuyển tiếp lưu lượng	45
3.5.6. Tổng quan luồng hoạt động của hệ thống	46
3.5.6.1. Tạo và quản lý container qua Rancher:	46
3.5.6.2. Docker Registry:	46
3.5.6.3. Cluster RKE2 với HA (High Availability):.....	46
3.5.6.4. Lưu trữ với Longhorn:	47
3.5.6.5. Ứng dụng và dịch vụ triển khai trên cluster:	47
3.5.6.6. Giám sát và cảnh báo:.....	48
Chương 4. Hiện thực hệ thống	49
4.1. Công cụ và môi trường	49
4.2. Triển khai Kubernetes	49
4.2.1. Triển khai RKE2 cluster bằng Ansible.....	49
4.2.1.1. Yêu cầu trước khi thực hiện.....	49

4.2.1.2. Triển khai Script	50
4.2.1.3. Giải thích Script	51
4.3. Triển khai các ứng dụng quản lí Cluster	64
4.3.1. Rancher	64
4.3.2. Monitoring:	66
4.3.3. Longhorn	67
4.3.4. Metallb.....	68
4.3.5. Docker Registry + UI	69
4.3.5.1. Docker Registry	69
4.3.5.2. Docker Registry UI	72
4.4. Triển khai các ứng dụng học máy trên Cluster.....	73
4.4.1. PostgreSQL.....	73
4.4.2. MinIO	75
4.4.3. MLflow	78
4.4.4. JupyterHub.....	80
Chương 5. Thực nghiệm và đánh giá	83
5.1. Kịch bản thực nghiệm.....	83
5.1.1. Triển khai và kiểm thử tính năng HA.....	83
5.1.2. Quản lý tài nguyên Cluster với Rancher	83
5.1.3. Quản lý lưu trữ với Longhorn	84
5.1.4. Triển khai và quản lý các tác vụ học máy bằng MLflow, MinIO và PostgreSQL	84
5.1.5. Triển khai học máy bằng JupyterHub	88
5.1.6. Giám sát và cảnh báo sử dụng Kube-Prometheus Stack	89

5.1.7.	Triển khai Docker Registry và UI	90
5.1.8.	Triển khai và kiểm thử MetalLB cho Load Balancing.....	90
5.2.	Kết quả thực nghiệm.....	91
5.2.1.	Đánh giá kết quả thực nghiệm	91
5.2.2.	So sánh với các công cụ/công trình liên quan	92
Chuong 6.	Kết luận và hướng phát triển.....	93
6.1.	Kết luận.....	93
6.2.	Hướng phát triển.....	94

DANH MỤC HÌNH

Hình 2.1: Kiến trúc của Kubernetes	8
Hình 2.2: Topology của mô hình Stacked Etcd.....	19
Hình 2.3: Topology của mô hình External Etcd.....	20
Hình 2.4: Thành phần và kiến trúc của RKE2.....	23
Hình 3.1: Kiến trúc của PostgreSQL, MinIO và MLflow	36
Hình 3.2: Kiến trúc của JupyterHub	36
Hình 3.3: Luồng triển khai RKE2 Cluster	37
Hình 3.4: RKE2 HA Topology với Keepalived	40
Hình 3.5: Sơ đồ minh họa cách Rancher hoạt động	41
Hình 3.6: Kiến trúc của Longhorn.....	43
Hình 3.7: Luồng hoạt động của Docker Registry	44
Hình 3.8: Luồng hoạt động của Metallb.....	45
Hình 3.9: Luồng hoạt động của hệ thống	46
Hình 4.1: Câu lệnh để triển khai RKE2 cluster	50
Hình 4.2: Giá trị mặc định	52
Hình 4.3: Câu lệnh cài đặt các gói cần thiết	52
Hình 4.4: Cài đặt SSH và sao chép khóa ssh.....	53
Hình 4.5: Cập nhật file /etc/hosts trên máy Ansible.....	54
Hình 4.6: Tạo file hosts cho Ansible Script	54
Hình 4.7: Lấy phiên bản RKE2 mới nhất	54
Hình 4.8: Triển khai RKE2 normal mode	55
Hình 4.9: Triển khai RKE2 HA mode	55
Hình 4.10: Vô hiệu hóa ufw	56
Hình 4.11: Cấu hình IP Forwarding	57
Hình 4.12: Tắt và vô hiệu hóa Swap	57
Hình 4.13: Cập nhật và nâng cấp hệ thống.....	58
Hình 4.14: Cập nhật file /etc/hosts và hostname	58

Hình 4.15: Cài đặt Open-iSCSI	59
Hình 4.16: Khởi động lại	59
Hình 4.17: File deploy_rke2.yaml	60
Hình 4.18: File deploy_rke2_ha.yaml	61
Hình 4.19: Tạo file /tmp/post_install.sh trên node master	62
Hình 4.20: Cập nhật và tải gói cần thiết	62
Hình 4.21: Kiểm tra và cài đặt curl	62
Hình 4.22: Cài đặt Helm	63
Hình 4.23: Cấu hình Ingress NGINX	63
Hình 4.24: Tạo symlink cho kubectl	64
Hình 4.25: Cấu hình biến môi trường KUBECONFIG	64
Hình 4.26: Đặt quyền, chạy và xóa script sau khi hoàn tất	64
Hình 4.27: Khai báo các biến, thêm và cập nhật repo, cài đặt hoặc nâng cấp cert-manager	65
Hình 4.28: Cài đặt hoặc nâng cấp Rancher với cấu hình đã chọn	66
Hình 4.29: Lệnh Helm cài đặt Kube-Prometheus-stack	66
Hình 4.30: Kết quả khi chạy lệnh helm	66
Hình 4.31: Cấu hình biến môi trường, thêm và cập nhật Helm repo	67
Hình 4.32: Tạo namespace và cài đặt Longhorn	68
Hình 4.33: Kiểm tra trạng thái Longhorn	68
Hình 4.34: Cài đặt và cấu hình Metallb	69
Hình 4.35: Tạo PVC cho Docker Registry	70
Hình 4.36: Thêm và update Helm repo	70
Hình 4.37: Cài đặt Docker Registry	70
Hình 4.38: Kiểm tra dịch vụ Docker Registry	71
Hình 4.39: Tải image Nginx mới nhất	71
Hình 4.40: Gắn tag cho image Nginx và push lên Registry	71
Hình 4.41: Pull Image Nginx về máy	71
Hình 4.42: Cấu hình daemon.json	72

Hình 4.43: Cấu hình registries.yaml trong RKE2	72
Hình 4.44: Giao diện của Docker Registry UI	73
Hình 4.45: Tạo namespace, PVC.....	74
Hình 4.46: Kiểm tra Deployment trong Rancher	74
Hình 4.47: Kiểm tra Service trong Rancher	74
Hình 4.48: Thêm Helm repo, tạo PVC	75
Hình 4.49: Tạo Secret	76
Hình 4.50: Trang đăng nhập của MinIO	77
Hình 4.51: Tạo Bucket.....	77
Hình 4.52: Tạo Access Key	78
Hình 4.53: Tạo Database trong PostgreSQL cho MLflow	78
Hình 4.54: Tạo và đẩy image của MLflow lên Docker Registry	79
Hình 4.55: Tạo PVC cho MLflow	79
Hình 4.56: Cấu hình để sử dụng image MLflow trên Docker Registry	80
Hình 4.57: Cấu hình PostgreSQL và s3 với tham số tương ứng trong hệ thống.	80
Hình 4.58: Cài đặt JupyterHub với Helm.....	81
Hình 4.59: Kết quả cài đặt JupyterHub với Helm	81
Hình 4.60: Trang đăng nhập JupyterHub	81
Hình 5.1: Theo dõi trạng thái Deployment của các công cụ	84
Hình 5.2: Kết nối đến PosgreSQL service và tạo Table.....	85
Hình 5.3: Câu lệnh để chạy ví dụ với MLflow.....	85
Hình 5.4: Kết quả của ví dụ trên CLI	85
Hình 5.5: Kết quả của Experiment trên MLflow UI	86
Hình 5.6: Object được cập nhật sau các lần sử dụng MLflow trên MinIO UI....	86
Hình 5.7: Chi tiết về các Artifacts	86
Hình 5.8: Thông tin Experiment.....	87
Hình 5.9: Thông Tin Chi Tiết Experiment	88
Hình 5.10: Biểu đồ hiển thị hiệu suất của các thí nghiệm hoặc các tham số	88
Hình 5.11: Trang chủ JupyterHub	89

- Hình 5.12: Docker Registry UI cập nhật các image được push90
Hình 5.13: Test thử tính năng cung cấp dịch vụ LoadBalancer của Metallb91

DANH MỤC BẢNG

Bảng 2.1: Bảng so sánh các tính năng của Docker và Kubernetes	25
Bảng 2.2: Bảng so sánh các loại Cluster	28

DANH MỤC TỪ VIẾT TẮT

AI - Trí tuệ nhân tạo (Artificial Intelligence)

ML - Học máy (Machine Learning)

K8s - Kubernetes

Docker - Một nền tảng container hóa ứng dụng

Pod - Đơn vị cơ bản trong Kubernetes, chứa một hoặc nhiều container

CNI - Container Network Interface

GPU - Đơn vị xử lý đồ họa (Graphics Processing Unit)

API - Giao diện lập trình ứng dụng (Application Programming Interface)

CI/CD - Liên tục tích hợp (Continuous Integration) và triển khai (Continuous Deployment)

CLI - Giao diện dòng lệnh (Command Line Interface)

DNS - Hệ thống tên miền (Domain Name System)

Ingress - Điều khiển định tuyến HTTP/HTTPS vào hệ thống Kubernetes

HPA - Autoscaling theo số lượng pod (Horizontal Pod Autoscaler)

VPA - Autoscaling theo tài nguyên yêu cầu (Vertical Pod Autoscaler)

Kubelet - Agent chạy trên mỗi node trong Kubernetes, đảm bảo các container hoạt động đúng cách

Kube-Proxy - Thành phần định tuyến lưu lượng mạng trong Kubernetes

Prometheus - Công cụ giám sát mã nguồn mở

Grafana - Công cụ hiển thị dữ liệu giám sát

TÓM TẮT

Trong bối cảnh học máy (Machine Learning - ML) ngày càng được ứng dụng rộng rãi trong các lĩnh vực như trí tuệ nhân tạo, phân tích dữ liệu và tự động hóa, việc điều phối container và quản lý tài nguyên hiệu quả trở thành một thách thức lớn để đảm bảo hiệu suất cao, tối ưu hóa sử dụng tài nguyên và giảm thời gian xử lý các tác vụ phức tạp. Nghiên cứu này tập trung vào việc khai thác Kubernetes (K8s) làm nền tảng chính để xây dựng hệ thống điều phối container và quản lý tài nguyên cho các tác vụ học máy. Mục tiêu của nghiên cứu là xây dựng một hệ thống toàn diện, tự động và hiệu quả, giúp tối ưu hóa quy trình xử lý học máy từ huấn luyện, kiểm thử đến triển khai, đồng thời đảm bảo tính sẵn sàng và khả năng mở rộng linh hoạt.

Phương pháp nghiên cứu bao gồm triển khai Kubernetes để điều phối các workload ML dưới dạng container, tận dụng khả năng tự động phân phối tài nguyên và quản lý quy trình làm việc của nó. Kubeflow được tích hợp như một bộ công cụ hỗ trợ các tác vụ ML đặc thù, bao gồm tiền xử lý dữ liệu, huấn luyện mô hình, đánh giá hiệu suất và triển khai mô hình. Để đáp ứng các yêu cầu tài nguyên lớn, hệ thống sử dụng các cơ chế autoscaling trên Kubernetes, cho phép tự động điều chỉnh tài nguyên (CPU, GPU, bộ nhớ) dựa trên khối lượng công việc và yêu cầu tính toán của các mô hình ML. Hơn nữa, Prometheus và Grafana được triển khai để giám sát toàn diện hiệu suất hệ thống, bao gồm trạng thái tài nguyên, hiệu suất container và thông tin chi tiết về workload ML. Để hỗ trợ lưu trữ và xử lý các tập dữ liệu lớn thường gặp trong ML, Longhorn được tích hợp làm giải pháp lưu trữ phân tán có khả năng chịu lỗi và đảm bảo hiệu năng cao.

Hệ thống được thử nghiệm trên nhiều loại workload ML khác nhau, từ các tác vụ nhỏ với yêu cầu tài nguyên tối thiểu đến các mô hình phức tạp đòi hỏi GPU chuyên dụng và khối lượng dữ liệu lớn. Kết quả cho thấy hệ thống đã thành công trong việc tự động phân bổ tài nguyên một cách linh hoạt, giảm thiểu thời gian xử lý trung bình cho các mô hình học máy và tối ưu hóa hiệu suất sử dụng phần cứng. Các công

cụ giám sát như Prometheus và Grafana không chỉ cung cấp thông tin chi tiết về tình trạng hệ thống mà còn giúp phát hiện và khắc phục các nút cốt chai (bottlenecks) trong việc phân phối tài nguyên. Việc sử dụng Longhorn đảm bảo khả năng lưu trữ hiệu quả và ổn định ngay cả trong môi trường có yêu cầu dữ liệu lớn và truy cập đồng thời.

Kết luận, hệ thống điều phối container và quản lý tài nguyên dựa trên Kubernetes được phát triển trong nghiên cứu này đã chứng minh khả năng đáp ứng các yêu cầu phức tạp của học máy, từ hiệu quả xử lý, tối ưu hóa tài nguyên đến đảm bảo tính sẵn sàng cao. Hệ thống không chỉ hỗ trợ mạnh mẽ cho các ứng dụng học máy mà còn mở ra tiềm năng triển khai trong các môi trường tính toán hiện đại với yêu cầu ngày càng cao về hiệu suất và linh hoạt.

Chương 1. Giới thiệu về tài

1.1. Giới thiệu vấn đề

Trong bối cảnh công nghệ hiện đại ngày nay, với sự phát triển mạnh mẽ của trí tuệ nhân tạo (AI) và học máy (Machine Learning), các tác vụ học máy ngày càng trở nên phức tạp và đòi hỏi tài nguyên tính toán ngày càng lớn. Các mô hình học sâu (Deep Learning) hiện nay cần xử lý lượng dữ liệu khổng lồ và yêu cầu tốc độ tính toán cao, dẫn đến việc sử dụng các công cụ và nền tảng mạnh mẽ để đáp ứng nhu cầu này là vô cùng quan trọng. Một trong những công nghệ đang trở thành xu hướng là Kubernetes, một hệ thống mã nguồn mở giúp tự động hóa việc triển khai, mở rộng, và quản lý các container.

1.1.1. Hiện trạng

Trong bối cảnh ngày càng gia tăng nhu cầu sử dụng học máy trong các lĩnh vực như phân tích dữ liệu, trí tuệ nhân tạo (AI), và Internet vạn vật (IoT), việc xây dựng hạ tầng xử lý tác vụ học máy hiệu quả đang trở thành một thách thức lớn. Hệ thống container, đặc biệt Kubernetes, được coi là giải pháp tiên tiến nhất nhằm điều phối và quản lý tài nguyên hiệu quả. Tuy nhiên, việc tích hợp học máy và Kubernetes đặt ra nhiều khó khăn, bao gồm:

- Quản lý tài nguyên linh hoạt để tối ưu hiệu năng trong môi trường phân tán.
- Giải quyết xung đột tài nguyên khi chạy đồng thời nhiều tác vụ.
- Tích hợp pipeline học máy một cách liên tục và tự động vào hệ thống container.
- Giảm thiểu chi phí và tối đa tái sử dụng tài nguyên.

1.1.2. Thách thức

Những khó khăn chính bao gồm:

- Tính linh hoạt và khả năng mở rộng: Việc phân bổ tài nguyên hiệu quả trong môi trường đa người dùng và đa tác vụ.

- Tính tự động và giám sát: Triển khai, giám sát, và quản lý pipeline học máy một cách tự động nhằm tăng hiệu suất.
- Hiệu năng: Tối ưu tính song song của tác vụ học máy trong môi trường Kubernetes, đảm bảo không gián đoạn.
- Khả năng bảo trì: Duy trì hệ thống đồng bộ và bền vững trong thời gian dài.

1.2. Mục tiêu của đồ án

- Xây dựng hệ thống quản lý và điều phối tài nguyên cho tác vụ học máy sử dụng Kubernetes một cách hiệu quả và tự động.
- Nghiên cứu giải pháp tối ưu phân bố tài nguyên nhằm tối đa hiệu quả xử lý, đặc biệt trong môi trường phân tán.
- Tích hợp pipeline học máy vào Kubernetes một cách hiệu quả, tự động, và linh hoạt.
- Tăng tính khả dẻ dự kiến cho hệ thống để xử lý các tác vụ phức tạp.

1.3. Phương pháp nghiên cứu

- Tìm hiểu tài liệu: Khảo sát và đánh giá các công nghệ hiện có như Kubernetes, Docker, và ML frameworks.
- Thiết kế và xây dựng: Phát triển các pipeline học máy, thiết lập quy trình triển khai trên hạ tầng Kubernetes.
- Thực nghiệm: Triển khai và đánh giá hệ thống qua các tình huống thực tế, đảm bảo khả thi và hiệu năng cao.
- Phân tích dữ liệu: Thu thập và đánh giá kết quả từ hệ thống, nhằm cải tiến hiệu quả xử lý.

Chương 2. Cơ sở lý thuyết

2.1. Container

2.1.1. Khái niệm

Container là một hình thức ảo hóa hệ điều hành, bên trong là các packages, một số dependencies. Dùng để giải quyết vấn đề chuyển giao phần mềm một cách đáng tin cậy giữa các môi trường khác nhau. Docker container image là một gói phần mềm nhẹ, chạy độc lập và có thể thực thi bao gồm mọi thứ để chạy ứng dụng như: code, runtime, system tools, system libraries và settings mà không bị các yếu tố môi trường hệ thống làm ảnh hưởng và ngược lại.

Công nghệ container, hay gọi đơn giản là container, là một phương pháp đóng gói ứng dụng để ứng dụng có thể chạy với các phụ thuộc của mình (gồm source code và library, runtime, framework...) một cách độc lập, tách biệt với các chương trình khác

2.1.2. Đặc điểm của Container

Các thành phần trong cấu trúc của Container gồm:

- Server: Là thành phần chính (có thể là máy chủ vật lý hoặc máy chủ ảo).
- Host OS: Hệ điều hành được cài đặt trên server.
- Container: Được đóng gói để chạy các ứng dụng có sự phụ thuộc đối với phần mềm và phần cứng.

Process trong mỗi container bị cô lập với các process của những container khác trong hệ thống. Tuy nhiên, tất cả container chia sẻ kernel của host O

2.2. Docker

2.2.1. Khái niệm

Docker hay Docker Engine là Container runtime (thành phần giúp chạy các ứng dụng dưới dạng Container) mã nguồn mở phổ biến cho phép phát triển, triển khai và

thử nghiệm các ứng dụng được Container hóa trên nhiều nền tảng khác nhau. Docker Container là các gói ứng dụng và tệp độc lập được tạo bằng khung Docker. Docker là nền tảng phần mềm cho phép bạn dựng, kiểm thử và triển khai ứng dụng một cách nhanh chóng. Docker đóng gói phần mềm vào các đơn vị tiêu chuẩn hóa được gọi là container có mọi thứ mà phần mềm cần để chạy, trong đó có thư viện, công cụ hệ thống, mã và thời gian chạy. Bằng cách sử dụng Docker, bạn có thể nhanh chóng triển khai và thay đổi quy mô ứng dụng vào bất kỳ môi trường nào và biết chắc rằng mã của bạn sẽ chạy được.

2.2.2. Kiến trúc của Docker

- Docker Engine: Cung cấp nền tảng cho việc quản lý container.
- Docker Images: Là các bản mẫu read-only chứa mọi thứ cần thiết để khởi chạy container.
- Docker Containers: Là các bản sao thực thi của Docker Image, độc lập với hệ thống máy chủ.
- Docker Hub: Kho lưu trữ các Docker Image, hỗ trợ chia sẻ và phân phối.

2.2.3. Quy trình thực thi

- Build

Đầu tiên tạo một dockerfile, trong dockerfile này chính là code của chúng ta. Dockerfile này sẽ được Build tại một máy tính đã cài đặt Docker Engine. Sau khi build ta sẽ có được Container, trong Container này chứa ứng dụng kèm bộ thư viện của chúng ta.

- Push

Sau khi có được Container, chúng ta thực hiện push Container này lên cloud và lưu tại đó.

- Pull, Run

Nếu một máy tính khác muốn sử dụng Container chúng ta thì bắt buộc máy phải thực hiện việc Pull container này về máy, tất nhiên máy này cũng phải cài Docker Engine. Sau đó thực hiện Run Container này.

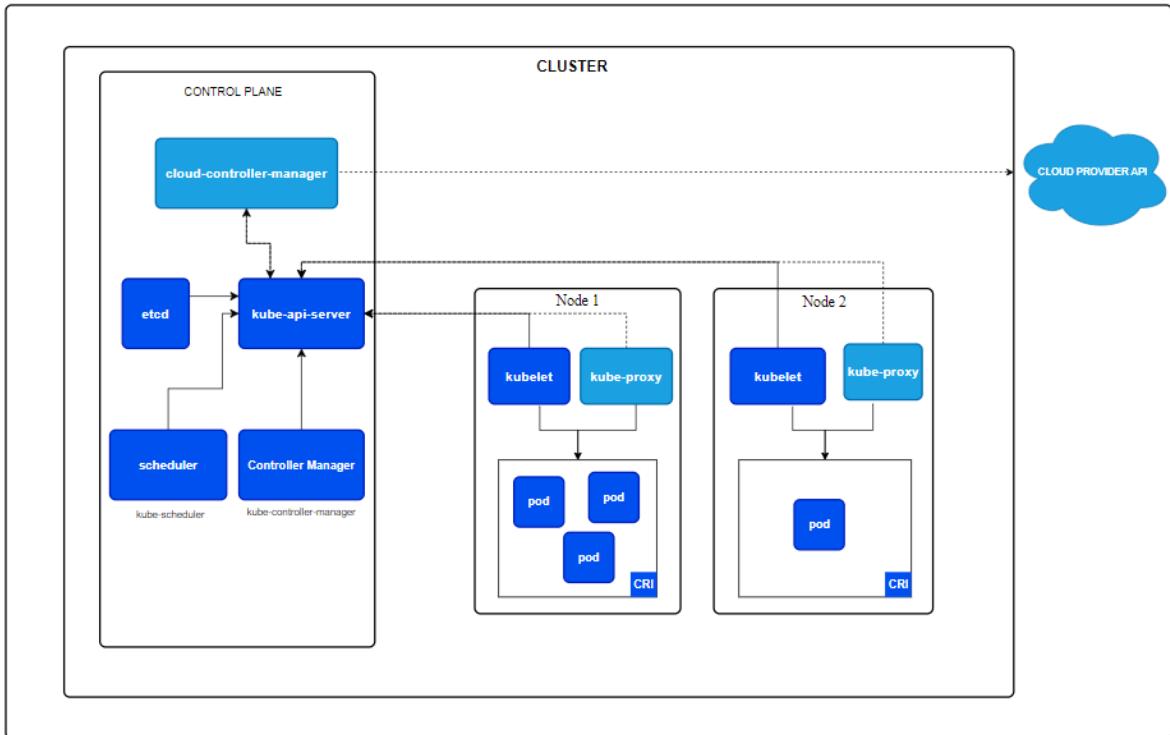
2.3. Kubernetes

Kubernetes, hoặc k8s là một nền tảng mã nguồn mở tự động hóa việc quản lý, scaling và triển khai ứng dụng dưới dạng container hay còn gọi là Container orchestration engine. Nó loại bỏ rất nhiều các quy trình thủ công liên quan đến việc triển khai và mở rộng các containerized applications [1].

Kubernetes orchestration cho phép bạn xây dựng các dịch vụ ứng dụng mở rộng nhiều containers. Nó lên lịch các containers đó trên một cụm, mở rộng các containers và quản lý tình trạng của các containers theo thời gian.

2.3.1. Kiến trúc chính

- Bộ điều khiển - The Master Node (hay còn gọi là Control Plane): Bao gồm API Server, Scheduler và Controller Manager. Đây là nơi quản lý và điều phối tất cả các hoạt động của cluster.
- The Worker Node - Nơi các pods được lên lịch: Bao gồm Kubelet, Kube-proxy và Container Runtime (thông thường là Docker). Đây là nơi chạy các container và thực hiện các yêu cầu từ Master Node.



Hình 2.1: Kiến trúc của Kubernetes

2.3.2. Các thành phần của Control Plane

Control Plane chịu trách nhiệm đưa ra các quyết định toàn cục cho cluster, như lên lịch Pods, duy trì trạng thái mong muốn của các ứng dụng, và quản lý vòng đời của các đối tượng trong Kubernetes. Các thành phần chính bao gồm:

2.3.2.1. Kube-API Server:

- Là công giao tiếp chính giữa các người dùng, công cụ, và các thành phần khác trong Kubernetes.
- API Server xử lý các yêu cầu RESTful và cung cấp giao diện để tương tác với cluster.
- Được thiết kế để mở rộng theo chiều ngang, có thể triển khai nhiều bản sao để cân bằng tải.

2.3.2.2. Etcd:

- Là cơ sở dữ liệu key-value phân tán, lưu trữ toàn bộ trạng thái của cluster.

- Đảm bảo tính nhất quán và độ tin cậy của dữ liệu trong cluster.

Lưu ý: Cần thiết lập kế hoạch sao lưu dữ liệu etcd để tránh mất mát dữ liệu quan trọng.

2.3.2.3. Kube-Scheduler:

- Là thành phần chịu trách nhiệm lên lịch các Pods tới các nút phù hợp dựa trên tài nguyên sẵn có, các yêu cầu cấu hình, và các ràng buộc khác.
- Các yếu tố xem xét khi lên lịch bao gồm:
 - Yêu cầu tài nguyên (CPU, RAM).
 - Ràng buộc phần cứng (GPU cho học máy).
 - Affinity/Anti-affinity (ưu tiên hoặc tránh đặt Pods gần nhau).
 - Locality dữ liệu (đặt Pods gần nguồn dữ liệu).

2.3.2.4. Kube-Controller-Manager

- Quản lý các bộ điều khiển khác nhau trong cluster, như:
- Node Controller: Theo dõi và phản ứng khi một node gặp sự cố.
- Replication Controller: Đảm bảo số lượng Pods mong muốn được chạy.
- Job Controller: Xử lý các tác vụ một lần (one-off tasks).
- EndpointSlice Controller: Liên kết các dịch vụ (Services) với Pods.

2.3.2.5. Cloud-Controller-Manager (Tùy chọn)

- Được sử dụng để tích hợp cluster với các API của nhà cung cấp dịch vụ đám mây (AWS, GCP, Azure).
- Không cần thiết nếu cluster được triển khai on-premise hoặc môi trường học tập.

2.3.3. Các thành phần của Worker Node

Worker Node thực hiện công việc chính của cluster bằng cách chạy các Pods. Mỗi node bao gồm:

2.3.3.1. Kubelet

- Là một agent chạy trên mỗi node, đảm bảo rằng các container trong Pod được triển khai và hoạt động đúng cách.
- Nhận chỉ thị từ API Server và quản lý trạng thái của Pods trên node đó.

2.3.3.2. Kube-Proxy

- Là thành phần chịu trách nhiệm định tuyến lưu lượng mạng giữa các Pods, cũng như giữa các Pods và các dịch vụ bên ngoài.
- Nếu bạn sử dụng plugin mạng (như Calico, Cilium), kube-proxy có thể được thay thế hoặc bổ sung.

2.3.3.3. Container Runtime

- Là thành phần quản lý vòng đời của container (chạy, dừng, và xóa container).
- Kubernetes hỗ trợ các container runtime như containerd, CRI-O, hoặc Docker (qua interface CRI).

2.3.4. Add-ons trong Kubernetes

2.3.4.1. DNS

DNS là một add-on quan trọng trong Kubernetes, cung cấp khả năng giải quyết tên miền cho các dịch vụ và Pod trong cụm. Khi các container cần tương tác với nhau, chúng sử dụng tên miền để liên lạc. DNS sẽ giải quyết các tên miền này thành địa chỉ IP của các container tương ứng, giúp chúng giao tiếp dễ dàng hơn. Các dịch vụ trong Kubernetes cũng được đăng ký trong DNS, cho phép các container truy cập dịch vụ bằng tên miền thay vì địa chỉ IP cụ thể.

2.3.4.2. Dashboard

Dashboard là một giao diện đồ họa giúp người dùng dễ dàng quản lý và giám sát trạng thái của hệ thống Kubernetes. Nó cung cấp giao diện trực quan để quản lý tài

nguyên và thông tin chi tiết về chúng. Dashboard cũng hỗ trợ quản lý tài khoản và xác thực đăng nhập, giúp người dùng theo dõi và quản lý hệ thống hiệu quả.

2.3.4.3. Ingress Controller

Ingress Controller là một add-on cung cấp chức năng định tuyến HTTP và HTTPS từ bên ngoài vào các dịch vụ trong cụm. Nó dựa trên các quy tắc định tuyến được cấu hình trong tài nguyên Ingress. Đây là thành phần quan trọng để triển khai các ứng dụng web trong Kubernetes.

2.3.4.4. Metrics Server

Metrics Server là một add-on quan trọng để giám sát hiệu suất của các container và tài nguyên trong hệ thống. Nó thu thập thông tin về CPU, RAM, băng thông mạng và các tài nguyên khác của container, sau đó hiển thị qua các báo cáo hoặc biểu đồ. Metrics Server hỗ trợ quản trị viên trong việc giám sát hiệu suất và giải quyết các vấn đề liên quan đến tài nguyên.

2.3.4.5. Network Plugins

Network plugins là các thành phần phần mềm thực hiện giao diện mạng container (CNI). Chúng chịu trách nhiệm cấp phát địa chỉ IP cho các Pod và cho phép chúng giao tiếp.

2.3.4.6. Service và Ingress

a) Service:

- Service là một đối tượng cốt lõi trong Kubernetes, cung cấp cách truy cập các Pod bằng một tên miền cố định. Service đảm bảo rằng các Pod có thể giao tiếp với nhau, ngay cả khi chúng bị thay đổi địa chỉ IP.
- Các loại Service:
 - ClusterIP: Chỉ khả dụng trong cụm.
 - NodePort: Mở cổng trên Node để truy cập từ bên ngoài.
 - LoadBalancer: Tích hợp với cân bằng tải đám mây.
 - ExternalName: Ánh xạ tới DNS bên ngoài.

- Ứng dụng: Service thường được sử dụng để định danh các nhóm Pod, cung cấp cân bằng tải và cho phép các dịch vụ giao tiếp dễ dàng hơn.

b) Ingress:

- Ingress định tuyến lưu lượng HTTP/HTTPS từ bên ngoài vào các dịch vụ trong Kubernetes dựa trên quy tắc tên miền hoặc đường dẫn.
- Lợi ích:
 - Giảm chi phí sử dụng IP công khai.
 - Cung cấp HTTPS và các tính năng cân bằng tải nâng cao.
- Ứng dụng: Ingress phù hợp cho các ứng dụng web phức tạp cần quản lý lưu lượng hiệu quả trong cụm. Các plugin mạng phổ biến bao gồm Calico, Flannel và Weave.

2.3.5. Workload trong Kubernetes

Trong Kubernetes, workload là một thuật ngữ dùng để chỉ các tác vụ hoặc ứng dụng mà bạn muốn chạy trên cluster Kubernetes. Workload trong Kubernetes không chỉ đơn thuần là các ứng dụng, mà còn có thể là các tác vụ tạm thời, các dịch vụ cần duy trì liên tục hoặc các ứng dụng phức tạp yêu cầu các cấu hình đặc biệt. Kubernetes cung cấp các đối tượng workload để bạn có thể quản lý, mở rộng và theo dõi các tác vụ của mình một cách hiệu quả.

2.3.5.1. Pod

- Pod là đơn vị cơ bản nhất trong Kubernetes. Một pod có thể chứa một hoặc nhiều container, các container này chia sẻ mạng và không gian lưu trữ tạm thời (volumes). Pod giúp bạn quản lý các container theo nhóm, đảm bảo rằng chúng luôn chạy cùng nhau và có thể giao tiếp dễ dàng với nhau.
- Mỗi pod trong Kubernetes sẽ có một địa chỉ IP riêng, và các container bên trong pod có thể giao tiếp với nhau thông qua các cổng mạng mà Kubernetes cung cấp.

- **Ứng dụng:** Pod thường được sử dụng khi bạn chỉ có một ứng dụng đơn giản hoặc một nhóm các ứng dụng phụ thuộc vào nhau và cần được chạy trong cùng một môi trường.

2.3.5.2. Deployment

- Deployment là đối tượng dùng để quản lý các bản sao (replicas) của pod. Deployment duy trì số lượng pod ổn định và tự động thay thế các pod bị lỗi. Deployment hỗ trợ các tính năng như rolling updates (cập nhật theo kiểu tuần tự) và rollback (quay lại phiên bản trước nếu có sự cố).
- **Ứng dụng:** Deployment thường được sử dụng cho các ứng dụng yêu cầu tính sẵn sàng cao, khả năng mở rộng linh hoạt và dễ dàng quản lý các bản sao (replicas). Deployment giúp bạn dễ dàng triển khai, cập nhật, quay lại các phiên bản ứng dụng.
- **Replicaset** là một đối tượng quan trọng giúp duy trì một số lượng bản sao (replicas) của pod ổn định trong cluster. Nó tự động tạo mới các pod khi cần thiết và đảm bảo rằng tổng số pod luôn duy trì ở mức đã được định cấu hình. ReplicaSet có thể hoạt động độc lập, nhưng trong thực tế, nó thường được sử dụng trong Deployment để quản lý và mở rộng các ứng dụng một cách hiệu quả.
 - Cấu trúc của ReplicaSet bao gồm ba phần: replicas xác định số lượng pod cần duy trì, selector là bộ lọc nhãn xác định pod quản lý, và template là mẫu pod dùng để tạo các bản sao mới.
 - ReplicaSet hoạt động bằng cách tạo số lượng pod ban đầu theo giá trị replicas, tự động tạo lại pod khi bị lỗi, và cho phép mở rộng hoặc thu hẹp số lượng pod bằng cách thay đổi giá trị replicas.

2.3.5.3. StatefulSet

- StatefulSet là đối tượng trong Kubernetes được thiết kế để quản lý các ứng dụng có trạng thái (stateful applications). Các ứng dụng này cần duy trì dữ

liệu và trạng thái giữa các lần khởi động lại hoặc khi pod bị di dời giữa các node trong cluster.

- **Ứng dụng:** StatefulSet thích hợp cho các ứng dụng cơ sở dữ liệu hoặc các dịch vụ yêu cầu lưu trữ và quản lý trạng thái, chẳng hạn như các hệ thống quản lý dữ liệu phân tán hoặc các ứng dụng cần một tên miền duy nhất. Ví dụ như Cassandra, MongoDB hoặc Zookeeper.

2.3.5.4. DaemonSet

- DaemonSet là đối tượng giúp đảm bảo rằng một pod sẽ chạy trên tất cả các node trong cluster (hoặc một nhóm node được chỉ định). Điều này rất hữu ích khi bạn cần triển khai các dịch vụ như log collection, monitoring, hoặc các tác vụ quản lý hệ thống cần phải chạy trên mọi node.
- **Ứng dụng:** DaemonSet thường được sử dụng cho các dịch vụ hệ thống như fluentd, prometheus-node-exporter, hoặc các agent giám sát cần chạy trên tất cả các node.

2.3.5.5. Job

- Job là đối tượng trong Kubernetes giúp thực hiện các tác vụ ngắn hạn hoặc cần thực thi một lần duy nhất và hoàn tất trước khi kết thúc. Job đảm bảo rằng một tác vụ sẽ được thực thi thành công, nếu thất bại, Job sẽ tự động thử lại cho đến khi thành công.
- **Ứng dụng:** Job thích hợp cho các tác vụ đơn lẻ như sao lưu dữ liệu, xử lý batch, hoặc nhập liệu.

2.3.5.6. CronJob

- CronJob là một dạng mở rộng của Job, dùng để thực hiện các tác vụ định kỳ, theo lịch trình giống như cron jobs trên hệ thống Unix.
- **Ứng dụng:** CronJob rất hữu ích khi bạn cần thực hiện các tác vụ định kỳ như sao lưu dữ liệu, gửi email báo cáo, hoặc thực hiện các tác vụ làm sạch hệ thống.

2.3.6. Storage trong Kubernetes

Trong Kubernetes, lưu trữ (storage) là một thành phần quan trọng giúp các ứng dụng container hóa có thể lưu trữ, truy cập và quản lý dữ liệu một cách bền vững. Dữ liệu của container thường bị mất khi container dừng hoặc bị xóa, do đó, Kubernetes cung cấp nhiều cơ chế lưu trữ để đáp ứng các nhu cầu lưu trữ dữ liệu tạm thời, lâu dài, hoặc phân tán.

2.3.6.1. Tổng quan

Kubernetes hỗ trợ nhiều loại storage để đáp ứng các yêu cầu khác nhau:

- Ephemeral Storage (Lưu trữ tạm thời): Dữ liệu chỉ tồn tại trong thời gian container hoặc Pod đang chạy. Khi container bị xóa, dữ liệu cũng sẽ bị mất.
- Persistent Storage (Lưu trữ lâu dài): Dữ liệu tồn tại ngoài vòng đời của container hoặc Pod, đảm bảo tính bền vững của dữ liệu.
- Distributed Storage (Lưu trữ phân tán): Dữ liệu được lưu trên nhiều node để tăng tính khả dụng và hiệu suất.

2.3.6.2. Các khái niệm quan trọng

a) Volume

- Volume trong Kubernetes là một không gian lưu trữ được gắn (mount) vào container trong Pod.
- Dữ liệu trong Volume tồn tại trong suốt vòng đời của Pod, nhưng sẽ mất nếu Pod bị xóa (trừ khi sử dụng Persistent Volume).
- Các loại Volume phổ biến:
 - emptyDir: Lưu trữ tạm thời, tạo một thư mục rỗng trong Pod. Dữ liệu sẽ mất khi Pod bị xóa.
 - hostPath: Sử dụng một thư mục trên máy chủ node. Không phù hợp cho các ứng dụng cần tính di động.
 - configMap: sử dụng để cung cấp thông tin cấu hình cho các pod.

- secret: Tương tự như ConfigMap, nhưng dùng để lưu trữ dữ liệu nhạy cảm, chẳng hạn như mật khẩu, token, hoặc chứng chỉ. Dữ liệu trong Secret được mã hóa và bảo mật.
- PersistentVolume (PV) và PersistentVolumeClaim (PVC): Dùng để quản lý lưu trữ lâu dài.

b) PersistentVolume (PV)

- PV là một tài nguyên lưu trữ trùu tượng được quản lý bởi cụm Kubernetes.
- PV tồn tại độc lập với Pod và có thể được sử dụng bởi nhiều Pod khác nhau.
- PV có thể tồn tại trong suốt vòng đời của cluster và có thể được sử dụng bởi bất kỳ pod nào thông qua một PersistentVolumeClaim (PVC).
- PV được định nghĩa trong cụm và liên kết với các dịch vụ lưu trữ như:
 - Bộ lưu trữ cục bộ (Local storage).
 - Bộ lưu trữ mạng (NFS, iSCSI).
 - Bộ lưu trữ đám mây (AWS EBS, Google Persistent Disk).

c) PersistentVolumeClaim (PVC)

- PVC là yêu cầu của người dùng đối với một PV cụ thể, bao gồm yêu cầu về dung lượng, loại lưu trữ (storage class), và chế độ truy cập.
- PVC định nghĩa kích thước lưu trữ và các yêu cầu về chế độ truy cập (read-write). Kubernetes sẽ tự động tìm kiếm một PV phù hợp với yêu cầu của PVC và liên kết chúng với nhau.

d) Cách hoạt động của PV và PVC

Cách hoạt động:

- Admin định nghĩa PV: Admin tạo các PV trong cụm với các thông số như dung lượng, loại lưu trữ, chế độ truy cập.
- Người dùng tạo PVC: Người dùng gửi yêu cầu lưu trữ bằng cách tạo PVC.
- Kubernetes gắn PVC với PV: Hệ thống tự động tìm PV phù hợp với yêu cầu từ PVC và thực hiện gắn kết.

Các chế độ truy cập (Access Modes):

- ReadWriteOnce (RWO): Volume chỉ có thể được gắn vào một node và cho phép đọc/ghi.
- ReadOnlyMany (ROX): Volume có thể được gắn vào nhiều node, nhưng chỉ ở chế độ đọc.
- ReadWriteMany (RWX): Volume có thể được gắn vào nhiều node và cho phép đọc/ghi.

e) StorageClass

- StorageClass định nghĩa cách Kubernetes cung cấp các PV.
- StorageClass giúp Kubernetes quyết định cách thức cấp phát lưu trữ và các thông số kỹ thuật của volume. StorageClass cung cấp các thông tin như loại bộ lưu trữ, vùng lưu trữ (zone), và các tham số cấu hình khác.
- Ví dụ: Với các dịch vụ đám mây như AWS, StorageClass có thể được định nghĩa để tự động tạo các EBS volumes.

2.3.7. Cách hoạt động của Kubernetes:

- Khai báo trạng thái mong muốn: Người dùng định nghĩa trạng thái mong muốn của hệ thống bằng cách sử dụng các đối tượng như Deployment, Service, và Pod trong file cấu hình YAML hoặc JSON.
- API Server xử lý yêu cầu: API Server nhận và xử lý các yêu cầu từ người dùng và các thành phần khác trong cluster.
- Scheduler quyết định vị trí: Scheduler quyết định nơi chạy các Pod trên các Worker Node dựa trên các yêu cầu tài nguyên và chiến lược phân tán.
- Kubelet triển khai và quản lý Pod: Kubelet quản lý việc triển khai các Pod trên các Worker Node và đảm bảo chúng chạy đúng cách.
- Kube-proxy quản lý mạng: Kube-proxy cung cấp dịch vụ mạng bằng cách cài đặt các quy tắc định tuyến và chuyển tiếp gói tin.

2.4. High Availability trong Kubernetes

High Availability (HA) trong Kubernetes là một yếu tố quan trọng để đảm bảo rằng ứng dụng của bạn luôn hoạt động ổn định và có thể truy cập được ngay cả khi xảy

ra sự cố hạ tầng hoặc bảo trì. Để đạt được HA trong Kubernetes, bạn cần áp dụng các chiến lược và kiến trúc nhằm đảm bảo rằng các thành phần của hệ thống (control plane, worker nodes, etcd) có khả năng phục hồi (resiliency) trước các lỗi hạ tầng và tránh downtime hoặc mất dữ liệu.

2.4.1. Các Thành Phần Chính trong HA

2.4.1.1. Master Node Redundancy:

- Vai trò: Các master node, hay control plane, chịu trách nhiệm quản lý toàn bộ cụm Kubernetes. Chúng điều phối các workload, quản lý trạng thái của cluster thông qua etcd và cung cấp API cho người dùng và các thành phần khác của hệ thống.
- Chiến lược triển khai:
 - Nhiều Master Nodes: Triển khai ít nhất ba master nodes để duy trì tính khả dụng cao. Việc sử dụng số lẻ cho phép đảm bảo quorum trong etcd, giúp cluster tiếp tục hoạt động ngay cả khi một node bị lỗi.
 - Phân phối địa lý: Đặt master nodes ở các khu vực hoặc vùng khác nhau để giảm thiểu rủi ro từ sự cố khu vực.
 - Tự động hóa với Kubeadm: Công cụ kubeadm giúp tự động hóa việc thiết lập control plane có tính HA. Kubeadm giúp sao chép các thành phần của control plane và triển khai các node master.

2.4.1.2. Worker Node Redundancy:

- Vai trò: Các worker nodes chạy các ứng dụng (pods) và khối lượng công việc của bạn. Đảm bảo rằng các worker nodes được phân phối và duy trì redundancy để tránh sự cố gián đoạn ứng dụng khi một node gặp sự cố.
- Chiến lược triển khai:
 - Triển khai đa vùng (Multi-Zone): Đặt các worker nodes ở nhiều vùng (zone) khác nhau trong cùng một region hoặc trên nhiều region khác nhau để đảm bảo rằng các workload có thể được di chuyển đến các node khỏe mạnh khác nếu một vùng gặp sự cố.

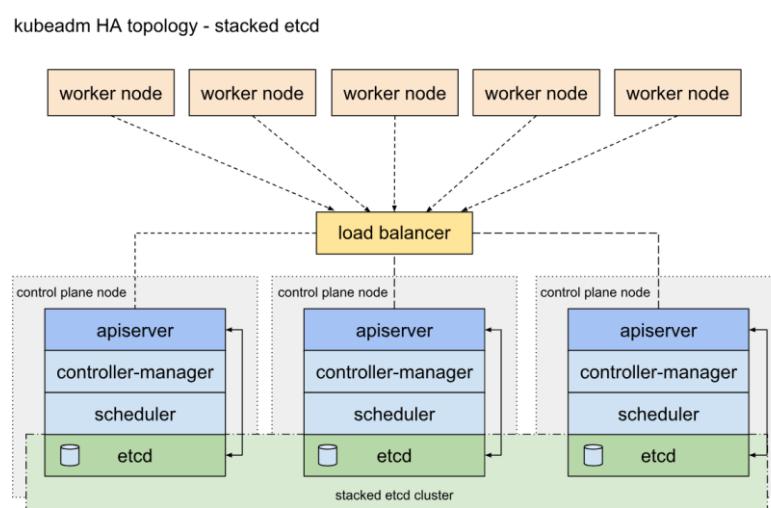
- Cân bằng tải (Load Balancing): Sử dụng load balancer để phân phối lưu lượng truy cập đều trên tất cả các worker nodes, tránh sự cố khi có điểm lỗi đơn lẻ.

2.4.1.3. Etcd Clustering:

- Vai trò: Etcd là một cơ sở dữ liệu phân tán key-value, lưu trữ tất cả dữ liệu cấu hình, trạng thái và metadata của cluster Kubernetes. Etcd đảm bảo tính đồng nhất và toàn vẹn dữ liệu của hệ thống.
- Chiến lược triển khai HA:
 - Clustered Mode: Đảm bảo rằng etcd chạy trên ít nhất ba node, giúp duy trì khả năng sẵn sàng ngay cả khi một node gặp sự cố.
 - Phân phối địa lý: Đặt các node etcd ở các khu vực khác nhau để bảo vệ khỏi các sự cố hạ tầng tại một khu vực cụ thể.

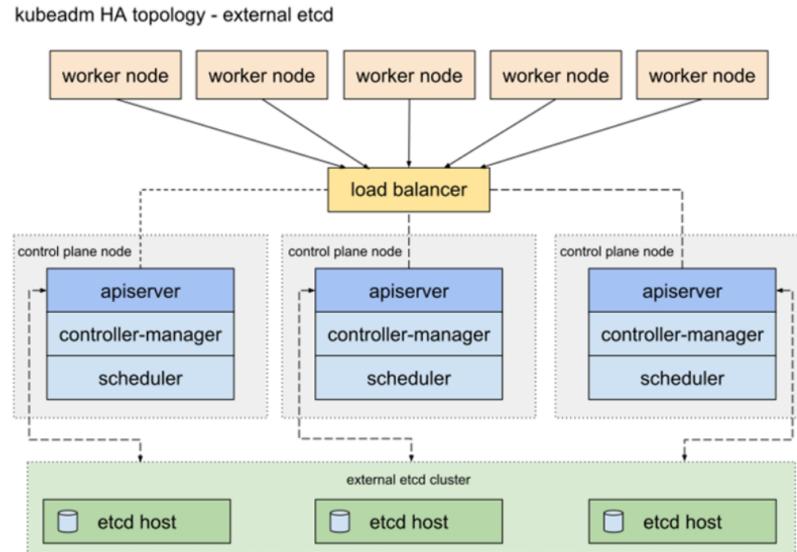
2.4.1.4. Topologies:

- Stacked Etcd Topology: Trong mô hình này, các thành phần control plane và etcd chạy trên cùng một node. Điều này dễ triển khai và quản lý, nhưng có rủi ro vì nếu một node gặp sự cố, cả control plane và etcd trên đó đều bị ảnh hưởng.



Hình 2.2: Topology của mô hình Stacked Etcd

- External Etc Topology: Trong mô hình này, các node etcd được triển khai trên các máy chủ riêng biệt, tách biệt với các node control plane. Điều này giúp tăng tính redundancy vì nếu một node control plane gặp sự cố, etcd vẫn có thể duy trì trạng thái của cụm.



Hình 2.3: Topology của mô hình External Etc

2.4.2. Các Thành Phần Cần Thiết để Đảm Bảo HA trong Kubernetes

- Control Plane Redundancy:

Quorum trong Etcd: Để đạt được tính nhất quán trong etcd, cần có ít nhất ba node etcd và một cấu hình quorum hợp lệ (số lẻ). Điều này giúp hệ thống vẫn có thể quyết định các thay đổi nếu có sự cố xảy ra.

- Node Redundancy:

- Master Nodes: Triển khai ít nhất ba master nodes để đảm bảo rằng hệ thống vẫn có thể duy trì khả năng quản lý và điều phối cluster ngay cả khi một node bị lỗi.
- Worker Nodes: Triển khai worker nodes trên các khu vực hoặc vùng khác nhau để đảm bảo rằng nếu một vùng hoặc một node gặp sự cố, các workload có thể được chuyển sang các node khác mà không ảnh hưởng đến ứng dụng.

- Cân Bằng Tải:

Load Balancer: Sử dụng load balancer để phân phối đều lưu lượng truy cập đến các master và worker nodes. Điều này tránh được điểm lỗi đơn lẻ và giúp tăng khả năng phục hồi của hệ thống khi có sự cố xảy ra.

2.4.3. Kỹ Thuật và Công Cụ Hỗ Trợ HA trong Kubernetes

2.4.3.1. Keepalived và Kube-VIP

a) Keepalived

- Vai trò: Keepalived là một công cụ cung cấp tính năng Virtual Router Redundancy Protocol (VRRP) giúp tạo ra một địa chỉ IP ảo (Virtual IP - VIP). VIP này sẽ được gán cho một trong các node trong cụm, và nếu node đó gặp sự cố, VIP sẽ tự động được chuyển sang node khác đang hoạt động.
- Cách hoạt động:
 - Keepalived sử dụng giao thức VRRP để giám sát trạng thái của các node và duy trì VIP.
 - Nếu node chính giữ VIP gặp sự cố, Keepalived sẽ chuyển VIP sang một node khác trong cụm, đảm bảo rằng các thành phần hoặc client có thể tiếp tục kết nối thông qua địa chỉ IP không đổi.

b) Kube-VIP

- Vai trò: Kube-VIP là một giải pháp triển khai Virtual IP được thiết kế đặc biệt cho các cụm Kubernetes. Nó đơn giản hóa việc cấu hình HA cho control plane mà không cần phải cài đặt thêm các công cụ phức tạp như Keepalived.
- Cách hoạt động:
 - Kube-VIP triển khai một dịch vụ DaemonSet chạy trên các node master.
 - Dịch vụ này chịu trách nhiệm duy trì VIP trên một trong các node master.
 - Kube-VIP tích hợp trực tiếp với Kubernetes thông qua API server và tự động chuyển VIP khi một node gặp sự cố.

2.4.3.2. Luồng Kết Nối với VIP và HA Control Plane

a) Kết Nối VIP với API Server:

- VIP được sử dụng như địa chỉ đầu cuối của API server trong Kubernetes. Mọi công cụ hoặc thành phần trong cluster (như kubelet, kubectl, hoặc các ứng dụng bên ngoài) khi cần truy cập API server sẽ kết nối tới VIP.
 - VIP được ánh xạ đến một trong các node master chạy API server.
- b) Luồng HA:
- Client gửi yêu cầu đến VIP: Tất cả các yêu cầu từ kubelet, kubectl, hoặc các dịch vụ bên ngoài đều được gửi tới địa chỉ VIP.
 - VIP chuyển tiếp yêu cầu đến API Server: VIP chuyển tiếp yêu cầu đến một node master đang giữ VIP. Node này chạy API server để xử lý yêu cầu.
 - Failover trong trường hợp lỗi: Nếu node master giữ VIP gặp sự cố, Keepalived hoặc Kube-VIP sẽ chuyển VIP sang một node master khác đang hoạt động, đảm bảo rằng kết nối không bị gián đoạn.

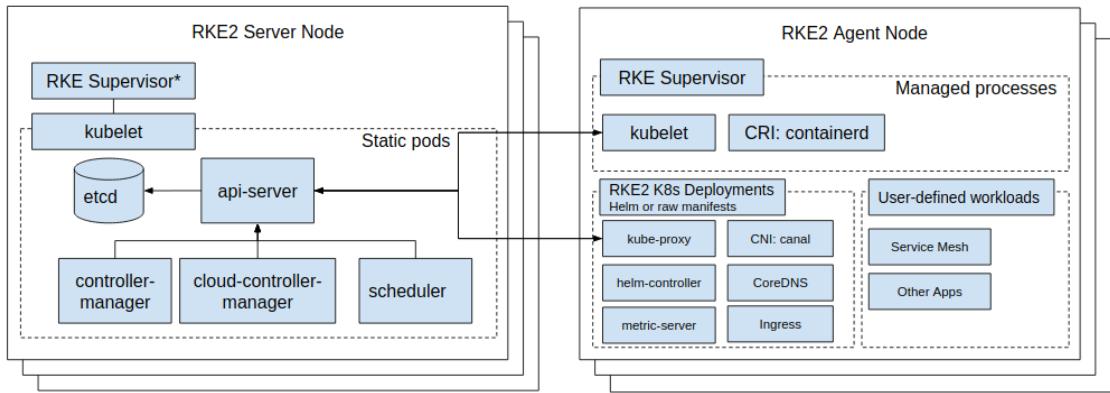
2.5. RKE2

RKE2, còn được gọi là RKE Government, là một phân phối Kubernetes thế hệ mới do Rancher Labs phát triển, hướng đến các yêu cầu bảo mật và tuân thủ trong lĩnh vực chính phủ Mỹ. RKE2 được thiết kế để cung cấp một nền tảng Kubernetes dễ sử dụng, hiệu suất cao, và mạnh mẽ cho các tổ chức cần đảm bảo tuân thủ các tiêu chuẩn nghiêm ngặt về bảo mật.

RKE2 được thiết kế với các tính năng bảo mật nâng cao, bao gồm:

- Tuân thủ CIS Benchmark: Cấu hình sẵn để đáp ứng tiêu chuẩn CIS Kubernetes Benchmark, giúp triển khai cụm Kubernetes an toàn mà không cần nhiều can thiệp từ quản trị viên.
- FIPS 140-2 Compliance: Hỗ trợ tiêu chuẩn mã hóa FIPS 140-2, đảm bảo hệ thống đáp ứng yêu cầu bảo mật liên bang.
- Quản lý bảo mật tích hợp: Tích hợp SELinux và Multi-Category Security (MCS) để kiểm soát quyền truy cập và áp dụng chính sách bảo mật nâng cao.
- Quét CVE định kỳ: Sử dụng Trivy để phát hiện lỗ hổng bảo mật thường xuyên trong các thành phần hệ thống.

2.5.1.1. Kiến trúc



Hình 2.4: Thành phần và kiến trúc của RKE2

Kiến trúc của RKE2 được xây dựng bao gồm 3 thành phần chính:

- Server (Control Plane): Control plane chịu trách nhiệm quản lý toàn bộ trạng thái của cluster. RKE2 server tương ứng với Kubernetes API server, controller manager và scheduler. Đây là nơi đưa ra các quyết định về lập lịch, điều phối trong Kubernetes.
- Agent (Worker Node): RKE2 agent node chịu trách nhiệm chạy khôi lượng công việc (container, pod, và service) được lập lịch trong cluster.
- Runtime (Containerd): RKE2 sử dụng containerd làm container runtime mặc định. Containerd là một container runtime chuẩn công nghiệp, nhẹ cho phép tạo ra các container. Containerd được xây dựng với mục đích cải thiện Kubernetes và tập trung vào bảo mật và điều phối container.

2.5.1.2. Các công nghệ được tích hợp sẵn

RKE2 là phiên bản nâng cao của Kubernetes, kết hợp nhiều công nghệ mã nguồn mở để cung cấp một hệ thống mạnh mẽ, bảo mật cao, phù hợp cho môi trường production. Các thành phần chính bao gồm:

- K3s: Helm Controller

- Kubernetes Core: API Server, Controller Manager, Kubelet, Scheduler, Proxy
- Các công nghệ hỗ trợ: etcd, runc, containerd/cri, CNI (Canal, Cilium, hoặc Calico), CoreDNS, Ingress NGINX Controller, Metrics Server, Helm

Ngoài NGINX Ingress Controller, các thành phần khác được biên dịch với Go+BoringCrypto để đảm bảo bảo mật cao, tuân thủ tiêu chuẩn mã hóa như FIPS 140-2.

RKE2 kế thừa kiến trúc Kubernetes, với khả năng mở rộng, khả dụng cao và chịu lỗi tốt. Mọi kết nối và giao tiếp trong cluster đều được mã hóa, làm cho RKE2 trở thành một lựa chọn lý tưởng cho môi trường sản xuất.

2.6. So sánh các công cụ

2.6.1. So sánh Docker và Kubernetes

Trong bối cảnh các hệ thống phần mềm hiện đại, việc ngày càng trở thành một thành phần quan trọng giúp đảm bảo tính ổn định, hiệu quả và khả năng mở rộng. Hai công nghệ phổ biến nhất trong lĩnh vực này là Docker và Kubernetes. Mặc dù cả hai đều liên quan đến container, nhưng chúng được thiết kế với các mục đích sử dụng khác nhau.

Tính năng	Docker	Kubernetes
Quản lý tài nguyên (CPU, RAM)	- Hỗ trợ giới hạn CPU và RAM thông qua các tham số <code>-cpus</code> , <code>--memory</code> .	- Hỗ trợ yêu cầu và giới hạn tài nguyên (requests và limits) để tối ưu hóa hiệu suất.
	- Docker Swarm cung cấp một số tính năng quản lý tài nguyên ở mức cụm.	- Quản lý tài nguyên chi tiết hơn, hỗ trợ phân bổ và tối ưu tài nguyên trên toàn cụm.
Quản lý mạng	- Cung cấp nhiều tùy chọn mạng: Bridge, Host, Overlay,	- Cung cấp các mô hình mạng phức tạp: Pod Network, Service

	Macvlan.	Network, Ingress.
	- Quản lý subnet, IP, và floating IP cơ bản.	- Hỗ trợ Network Policies để kiểm soát lưu lượng mạng giữa các Pod và dịch vụ.
Lập lịch tài nguyên	- Không hỗ trợ lập lịch trực tiếp. Docker Swarm chỉ hỗ trợ lập lịch ở mức cơ bản.	- Hỗ trợ hệ thống lập lịch mạnh mẽ dựa trên yêu cầu tài nguyên và các ràng buộc như affinity/anti-affinity.
High Availability (HA)	<ul style="list-style-type: none"> - Hỗ trợ HA thông qua Docker Swarm. Docker tự động quản lý và khởi động lại container khi có sự cố. 	<ul style="list-style-type: none"> - Được thiết kế để hỗ trợ HA và clustering từ đầu. Hỗ trợ nhiều master node để đảm bảo tính ổn định.
Khả năng mở rộng (Scalability)	<ul style="list-style-type: none"> - Docker Swarm hỗ trợ mở rộng nhưng không hiệu quả bằng Kubernetes. 	<ul style="list-style-type: none"> - Kubernetes hỗ trợ tự động mở rộng dựa trên tài nguyên hoặc tải công việc (Horizontal Pod Autoscaler).

Bảng 2.1: Bảng so sánh các tính năng của Docker và Kubernetes

Docker phù hợp để phát triển và chạy container trên môi trường nhỏ gọn, trong khi Kubernetes là lựa chọn hàng đầu cho hệ thống lớn cần quản lý nhiều container trên một cụm lớn với các yêu cầu phức tạp như tự động mở rộng, quản lý tài nguyên chi tiết, và High Availability. Trong bối cảnh đó án này, Kubernetes sẽ được lựa chọn làm công cụ chính để điều phối container và quản lý tài nguyên, đảm bảo tối ưu hiệu suất và độ tin cậy cho các tác vụ học máy.

2.6.2. So sánh các loại cluster

Trong quá trình nghiên cứu và triển khai các công cụ cho đồ án, tôi đã tìm hiểu và so sánh các loại Kubernetes cluster như đã đề cập ở mục 3.2.1 để đánh giá hiệu quả và phù hợp với mục tiêu của đồ án. Bảng dưới đây trình bày sự so sánh giữa các loại cluster như Vanilla Kubernetes, Minikube, MicroK8s, K3s, RKE và RKE2, từ đó giúp lựa chọn giải pháp phù hợp nhất cho việc triển khai, quản lý tài nguyên và tự động hóa các tác vụ học máy trong môi trường Kubernetes.

Tiêu chí	Vanilla Kubernetes Cluster	Minikube	MicroK8s	K3s	RKE	RKE2
Mục đích	Triển khai linh hoạt trên mọi môi trường, quy mô lớn	Phát triển, thử nghiệm cục bộ	Cài đặt nhanh cho cục bộ hoặc sản xuất nhỏ	Tối ưu cho edge, IoT và môi trường nhẹ	Phân phối đơn giản cho môi trường container	Phân phối bảo mật, tuân thủ môi trường doanh nghiệp
Nhà phát triển	CNCF	Kubernetes SIGs	Canonical (Ubuntu)	Rancher Labs	Rancher Labs	Rancher Labs
Container Runtime	containerd, Docker, CRI-O	Docker	containerd	containerd	Docker	containerd
Cài đặt	Phức tạp, đòi hỏi nhiều bước	Dễ dàng với một lệnh	Cài đặt nhanh qua Snap	Tập lệnh đơn giản hoặc Docker	Tập lệnh đơn giản	Tập lệnh đơn giản
Yêu cầu tài nguyên	Cao, tùy thuộc vào	Cao (yêu cầu Docker)	Thấp, nhẹ nhàng	Rất thấp, tối ưu cho	Trung bình	Trung bình, tối

	cấu hình hoặc VM)			hiệu suất		ưu cho bảo mật
Cấu hình cluster	Linh hoạt, triển khai đa dạng	Cluster đơn node cục bộ	Cluster đa node, mở rộng dễ dàng	Hỗ trợ cluster HA, phù hợp môi trường phân tán	Cluster đa node	Cluster HA với static pods
Khả năng mở rộng	Rất tốt, dùng cho hệ thống lớn	Hạn chế, chủ yếu thử nghiệm	Tốt, mở rộng đến cluster lớn	Tốt cho triển khai nhỏ và edge	Tốt, nhưng phụ thuộc Docker	Rất tốt, hỗ trợ đa môi trường
Tính năng bảo mật	Phụ thuộc vào quản trị viên	Hạn chế bảo mật	Hỗ trợ SELinux và AppArmor	Một số tính năng bảo mật	Cơ bản	Tích hợp CIS, FIPS 140-2, SELinux
Tuân thủ Upstream	Hoàn toàn tuân thủ	Tuân thủ nhưng đơn giản hóa	Hoàn toàn tuân thủ	Không hoàn toàn tuân thủ	Tuân thủ nhưng không nghiêm ngặt	Hoàn toàn tuân thủ
Hỗ trợ Add-on	Tích hợp nhiều addon như Helm, Ingress, Dashboard	Dashboard, Metrics-server, DNS	Istio, Helm, DNS, nhiều addon khác	Ingress, Helm, Load Balancer	Addons cơ bản	Addons nâng cao cho doanh nghiệp
High	Tùy chỉnh	Không hỗ	Hỗ trợ	Hỗ trợ	Hỗ trợ,	Tích hợp

Availability (HA)		trợ nhưng hạn chế	multi-node và HA	nhưng phức tạp hơn RKE2	sẵn với static pods	
Hỗ trợ môi trường	Linux, Windows, macOS	Linux, Windows, macOS	Ubuntu (tối ưu), Windows, macOS	Linux (hỗ trợ ARM, x86), Windows qua WSL2	Linux, Windows	Linux (tối ưu ARM/x86, air-gapped)

Bảng 2.2: Bảng so sánh các loại Cluster

Sau khi đánh giá khả năng mở rộng, bảo mật, linh hoạt triển khai và hỗ trợ môi trường, tôi quyết định sử dụng RKE2 cho đồ án. RKE2 cung cấp khả năng mở rộng linh hoạt, bảo mật cao với các tính năng như CIS, FIPS 140-2, SELinux, và hỗ trợ High Availability (HA) cùng cấu hình cluster đa node. Những đặc điểm này giúp tôi quản lý và duy trì hệ thống Kubernetes hiệu quả trong môi trường phân tán, đáp ứng tốt các yêu cầu của đồ án về quản lý tài nguyên và tự động hóa tác vụ học máy.

Chương 3. Phân tích thiết kế hệ thống

3.1. Vấn đề

Trong môi trường học máy (Machine Learning), các tác vụ tính toán thường yêu cầu tài nguyên phần cứng lớn và hiệu suất cao, như CPU, GPU, và bộ nhớ RAM. Việc triển khai các mô hình học máy đòi hỏi sự linh hoạt trong việc phân bổ tài nguyên và tự động hóa trong việc quản lý các tác vụ này, đặc biệt khi làm việc với lượng dữ liệu lớn. Điều này dẫn đến nhu cầu về một hệ thống quản lý container mạnh mẽ, có khả năng điều phối và giám sát tài nguyên hiệu quả.

Vấn đề cần giải quyết:

- Quản lý tài nguyên cho các tác vụ học máy: Làm thế nào để phân bổ và giám sát tài nguyên (CPU, RAM, bộ nhớ) cho các container chạy các tác vụ học máy?
- Điều phối container: Làm thế nào để triển khai và quản lý các container chạy các tác vụ học máy sao cho hiệu quả nhất, đặc biệt khi có sự thay đổi về tài nguyên hoặc yêu cầu tính toán?
- Tính linh hoạt và tự động hóa: Cần phải tự động hóa quy trình triển khai, điều phối, và giám sát các container để hỗ trợ việc phát triển, huấn luyện, và triển khai mô hình học máy.

3.2. Phương pháp

3.2.1. Tìm hiểu các tính năng của Docker và Kubernetes

- Vấn đề: Trước khi triển khai hệ thống container, cần phải hiểu rõ các công cụ Docker và Kubernetes, cùng những điểm mạnh và yếu của chúng.
- Phương pháp:
 - Nghiên cứu về Docker và Kubernetes, đặc biệt là các tính năng của từng công cụ.

- So sánh Docker và Kubernetes về các yếu tố như khả năng quản lý container, tính linh hoạt trong việc mở rộng, và khả năng cung cấp các dịch vụ cao cấp như scaling và orchestration như trong bảng 2.1.

3.2.2. Cài đặt Docker và Kubernetes

- Vấn đề: Cần thiết lập môi trường cơ sở hạ tầng để triển khai các ứng dụng container.
- Phương pháp:
 - Cài đặt Docker và các công cụ giám sát tài nguyên như CTOP để theo dõi hoạt động của Docker.
 - Triển khai Kubernetes bằng cách sử dụng kubeadm và các công cụ khác như Minikube, K3s, Microk8s để có thể so sánh các loại cài đặt Kubernetes khác nhau như trong bảng 2.2.
- Từ đó đưa ra quyết định về cấu hình cluster phù hợp cho các tác vụ học máy.

Cuối cùng, tôi chọn Kubernetes là công cụ phù hợp nhất cho đồ án vì khả năng mở rộng và tự động hóa, kết hợp với Docker để đóng gói các mô hình học máy và dữ liệu vào container.

3.2.3. Tự động hóa việc tạo Cluster và khám phá các tính năng

- Vấn đề: Việc thiết lập cluster Kubernetes phức tạp và cần được tự động hóa để tiết kiệm thời gian.
- Phương pháp:
 - Viết Ansible Script tự động hóa việc tạo cluster.
 - Cài đặt cluster bằng Kubespray để triển khai một môi trường Kubernetes hoàn chỉnh.
 - Tìm hiểu các tính năng như Authentication, Authorization, High Availability (HA), và quản lý tài nguyên.

3.2.4. Cài đặt và khởi tạo RKE, RKE2 Cluster

- Vấn đề: Cần thiết lập môi trường với khả năng quản lý mạnh mẽ, có thể mở rộng và dễ dàng quản lý.

- Phương pháp:
 - Tìm hiểu về RKE (Rancher Kubernetes Engine) và RKE2, chọn RKE2 vì tính năng mạnh mẽ và tối ưu hóa cho môi trường sản xuất[2].
 - Sử dụng Ansible Script để tự động hóa việc triển khai RKE2.
 - Cài đặt Rancher qua Helm để quản lý các cluster Kubernetes và kiểm soát chúng dễ dàng hơn.
 - So sánh RKE1 và RKE2 để lựa chọn công cụ phù hợp nhất theo nhu cầu 2.2.

3.2.5. Tìm hiểu về Rancher và các công cụ tích hợp

- Vấn đề: Rancher là một công cụ mạnh mẽ để quản lý các cluster Kubernetes, nhưng cần thử nghiệm các công cụ tích hợp sẵn để hiểu rõ hơn về cách sử dụng.
- Phương pháp:
 - Cài đặt Longhorn cho storage và sử dụng các công cụ Monitoring như Prometheus + Grafana để theo dõi và giám sát hiệu suất hệ thống.
 - Tạo cluster bằng Ansible và Bash Scripts: Sử dụng Ansible script và Bash script để tự động hóa quá trình cài đặt VM trước và sau khi cài đặt RKE2.
 - Kiểm tra các tính năng của Rancher và cách chúng hỗ trợ các cluster Kubernetes.

3.2.6. Cài đặt High Availability (HA) cho Master Node

- Vấn đề: Đảm bảo tính khả dụng cao cho master nodes để tránh sự cố mất kết nối hoặc downtime của hệ thống.
- Phương pháp:
 - Cài đặt HA cho master node trong cluster với cấu hình gồm 3 master nodes và 2 worker nodes.
 - Tìm hiểu cách thức triển khai HA và các phương pháp bảo đảm tính khả dụng trong Kubernetes.

3.2.7. Cài đặt và cấu hình LDAP Server

- Vấn đề: Tạo hệ thống xác thực người dùng và quản lý quyền truy cập cho các cluster Kubernetes.
- Phương pháp:
 - Cài đặt LDAP Server và cấu hình tích hợp với Rancher để quản lý người dùng và nhóm người dùng.
 - Ban đầu, tôi đã cài đặt OpenLDAP trên Ubuntu để tạo một external provider để kết nối với Rancher, tuy nhiên gặp phải sự cố khi kết nối không thành công.
 - Sau đó, tôi đã chuyển sang sử dụng OpenLDAP qua Helm, điều này giúp kết nối thành công, nhưng cuối cùng tôi quyết định sử dụng hệ thống xác thực và phân quyền người dùng có sẵn của Rancher để đơn giản hóa quản lý.
 - Tìm hiểu và cấu hình các tính năng quản lý tài nguyên như CPU và RAM thông qua request và limit trong các pod.
 - Tìm hiểu và triển khai Horizontal Pod Autoscaler (HPA) và Vertical Pod Autoscaler (VPA) để tự động điều chỉnh tài nguyên.

3.2.8. Cài đặt các công cụ DevOps

- Vấn đề: Cần các công cụ để tự động hóa quy trình CI/CD, theo dõi hệ thống, và triển khai ứng dụng.
- Phương pháp:
 - Cài đặt ArgoCD để quản lý việc triển khai ứng dụng theo cách GitOps.
 - Cài đặt các công cụ monitoring như Prometheus và Grafana để theo dõi hoạt động của các dịch vụ.
 - Cài đặt MetalLB cho load balancing và hỗ trợ mạng.
 - Cài đặt Docker Registry và các công cụ liên quan để quản lý container images.

- Cài đặt PostgreSQL, MinIO và MLflow để lưu trữ dữ liệu và hỗ trợ các tác vụ machine learning.
- Cài đặt JupyterHub cho môi trường làm việc và phát triển dữ liệu.

3.2.9. Tổng kết phương pháp

Phương pháp thực hiện dự án tập trung vào nghiên cứu, thử nghiệm và áp dụng các công cụ hiện đại trong quản lý container và điều phối tài nguyên. Kubernetes được chọn làm nền tảng chính nhờ khả năng mở rộng và tự động hóa. Các công cụ như RKE2, Rancher, Prometheus, Grafana, MetalLB, Longhorn được triển khai để đảm bảo tính khả dụng, tối ưu hóa mạng, lưu trữ và bảo mật. Việc tự động hóa bằng Ansible và Bash Scripts giúp tiết kiệm thời gian và giảm sai sót. Đồng thời, các công cụ DevOps như MLflow và JupyterHub hỗ trợ quy trình học máy và phát triển ứng dụng. Dự án thành công xây dựng một môi trường hiệu quả, mạnh mẽ, và tiết kiệm chi phí với các giải pháp mã nguồn mở.

3.3. Kiến trúc

3.3.1. Kiến trúc triển khai RKE2 Cluster

Để đáp ứng nhu cầu điều phối container và quản lý tài nguyên trong dự án “Điều phối Container và Quản lý Tài nguyên cho các Tác vụ Học máy với Kubernetes”, hạ tầng RKE2 đã được thiết kế và triển khai như sau:

3.3.1.1. Ansible Control Machine

- Vai trò: Ansible Control Machine đóng vai trò là máy điều khiển trung tâm, được sử dụng để chạy các script tự động hóa quá trình cài đặt và triển khai cluster RKE2 trên các máy ảo (VM). Nhờ vào Ansible, các nhiệm vụ triển khai trở nên dễ dàng, nhanh chóng và đảm bảo đồng bộ.
- Hệ điều hành: Linux (Ubuntu)
- Chức năng chi tiết:
 - Chạy các tập lệnh để tự động hóa việc cài đặt RKE2 trên các máy chủ trong cụm.

- Kết nối tới các máy ảo (VM) thông qua giao thức SSH để thực hiện các tác vụ từ xa như: Cài đặt các gói cần thiết; quản lý toàn bộ quá trình triển khai và ghi nhận các lỗi (nếu có).

3.3.1.2. Virtual Machine (VM)

Hệ tầng cluster bao gồm các VM được sắp xếp như sau:

a) Master Nodes

Chức năng:

- Đảm bảo High Availability (HA) nhờ vào triển khai RKE2 HA.
- Chạy các dịch vụ quản lý control plane bao gồm API Server, Scheduler, và Controller Manager.
- Phân phát tài nguyên và điều phối workloads trên các worker nodes.

Mạng: Có cài đặt SSH để kết nối với Ansible Control Machine.

b) Worker Nodes

Chức năng:

- Xử lý workloads như các pod chạy MLflow, JupyterHub, v.v.
- Làm việc với Persistent Volume (PV) từ Longhorn để lưu trữ dữ liệu.

Mạng: Có cài đặt SSH để kết nối với Ansible Control Machine.

3.3.2. Kiến trúc triển khai các ứng dụng quản lí cluster

Hệ thống được xây dựng với mục tiêu điều phối container và quản lý tài nguyên cho các tác vụ học máy, sử dụng Kubernetes làm nền tảng chính. Các thành phần chính của kiến trúc được thiết kế để đảm bảo tính sẵn sàng cao (HA), quản lý hiệu quả tài nguyên, và hỗ trợ các tác vụ học máy thông qua các công cụ và dịch vụ tích hợp.

3.3.2.1. Kubernetes Cluster

- RKE2 (Rancher Kubernetes Engine 2) được sử dụng để triển khai và quản lý cluster Kubernetes. Cluster này bao gồm 3 Master Nodes (được cấu hình với HA - High Availability) và 2 Worker Nodes. Mỗi Master Node có thể chịu

trách nhiệm cho một phần của hệ thống, giúp hệ thống đạt được tính sẵn sàng cao và khả năng chịu lỗi.

- Canal CNI được sử dụng để quản lý kết nối mạng giữa các pod trong cluster, cung cấp khả năng giao tiếp an toàn và hiệu quả.
- Để đảm bảo HA cho API server và các thành phần khác của control plane, hệ thống sử dụng Keepalived hoặc Kube-vip.

3.3.2.2. Quản lý Tài Nguyên

- Longhorn được sử dụng để cung cấp lưu trữ phân tán cho các ứng dụng, đặc biệt là cho các tác vụ học máy yêu cầu lưu trữ dữ liệu lớn và độ bền cao.
- MetalLB cung cấp khả năng load balancing cho các dịch vụ trong cluster Kubernetes, đặc biệt hữu ích trong các môi trường không sử dụng cloud.

3.3.2.3. Quản lý Docker Images (Docker Registry và UI)

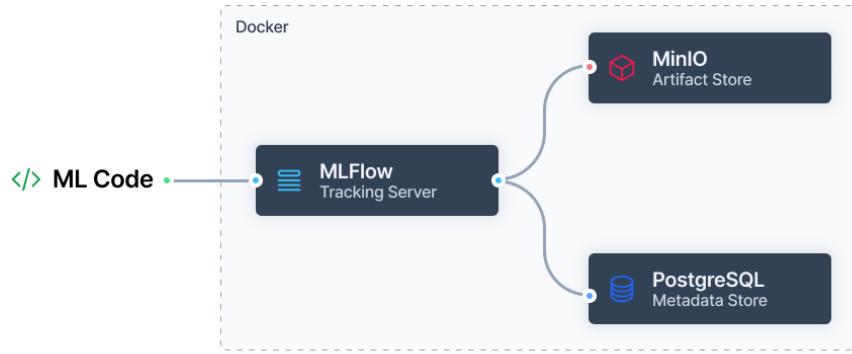
Docker Registry được sử dụng để lưu trữ các image Docker, bao gồm các image của mô hình học máy. UI của Docker Registry giúp quản lý và theo dõi các image đã được lưu trữ, với các tính năng như tìm kiếm, tải lên, tải xuống, v.v.

3.3.2.4. Giám Sát và Logging (Kube-Prometheus Stack)

- Prometheus thu thập và giám sát các chỉ số từ các container và pod trong cluster Kubernetes.
- Grafana cung cấp các dashboard trực quan để theo dõi hiệu suất và tình trạng của các dịch vụ.
- Kube-Prometheus Stack bao gồm các công cụ như Prometheus, Grafana, và Alertmanager giúp giám sát cluster và ứng dụng Kubernetes.

3.3.3. Kiến trúc triển khai các ứng dụng học máy trong cluster

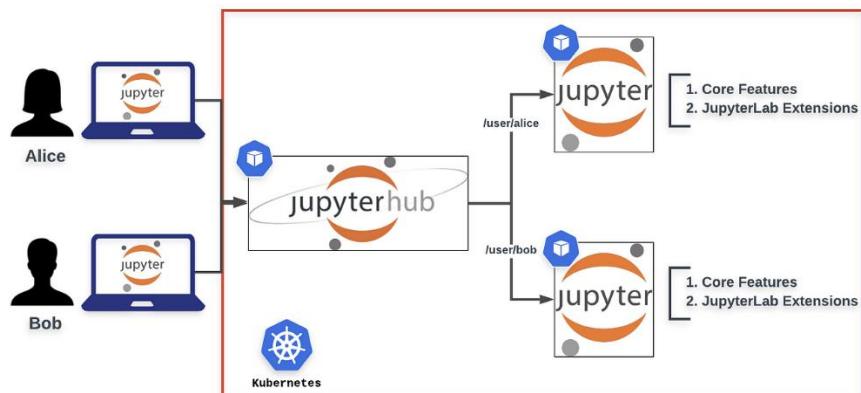
3.3.3.1. Quản lý và Triển khai Mô Hình Học Máy



Hình 3.1: Kiến trúc của PostgreSQL, MinIO và MLflow

- MLflow là công cụ quản lý vòng đời mô hình học máy (Machine Learning Lifecycle Management). MLflow giúp theo dõi các thử nghiệm, quản lý mô hình, và triển khai mô hình.
- PostgreSQL lưu trữ các metadata liên quan đến các mô hình học máy như các tham số, kết quả huấn luyện, v.v.
- MinIO là một giải pháp lưu trữ đối tượng, tương thích với S3, được sử dụng để lưu trữ các artifacts như mô hình học máy, dữ liệu huấn luyện và các tệp liên quan.

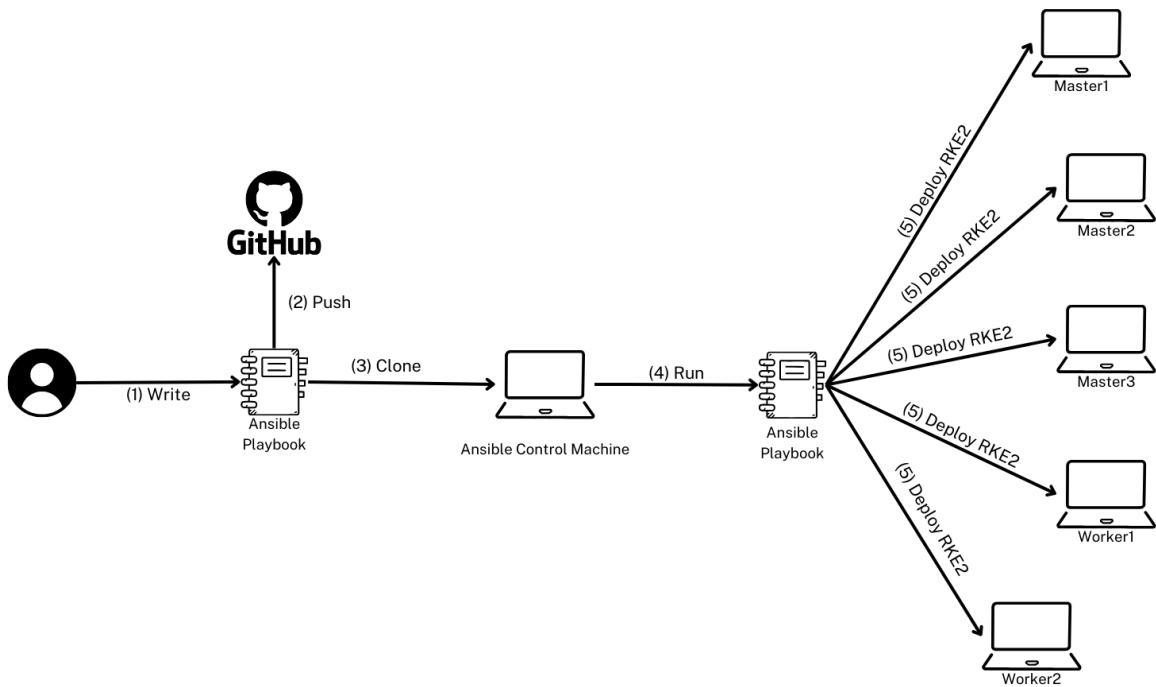
3.3.3.2. Quản lý Môi Trường Làm Việc (JupyterHub)



Hình 3.2: Kiến trúc của JupyterHub

JupyterHub cung cấp môi trường làm việc cho các nhà phát triển, nhà nghiên cứu và các kỹ sư dữ liệu, cho phép họ thực hiện các tác vụ học máy trong các Jupyter Notebooks. Các notebook có thể được sử dụng để huấn luyện mô hình học máy và triển khai các mô hình lên Kubernetes.

3.4. Luồng triển khai RKE2 Cluster



Hình 3.3: Luồng triển khai RKE2 Cluster

Luồng triển khai RKE2 Cluster được thiết kế để tự động hóa toàn bộ quá trình cài đặt và cấu hình cụm RKE2. Các bước thực hiện được mô tả chi tiết dưới đây, cùng với sơ đồ minh họa để làm rõ các giai đoạn triển khai.

Mô tả chi tiết các bước triển khai

3.4.1. Viết Ansible Playbook

- Người dùng viết các tập lệnh (playbooks) bằng Ansible để tự động hóa việc cài đặt và cấu hình cụm RKE2 [3].
- Các Playbook được cấu trúc thành các nhiệm vụ (tasks) để thực hiện các hành động như:

- Cài đặt các gói cần thiết (Docker, containerd).
- Cài đặt RKE2 trên các master và worker node.
- Thiết lập cấu hình mạng.
- Tải một vài công cụ cần thiết như helm lên master node.

3.4.2. Push Ansible Playbook lên GitHub

- Sau khi hoàn thành, Ansible Playbook được đẩy (push) lên một kho lưu trữ GitHub.
- Điều này đảm bảo rằng mã nguồn được lưu trữ an toàn, dễ dàng chia sẻ và quản lý phiên bản.

3.4.3. Clone Ansible Playbook về Ansible Control Machine

- Từ máy điều khiển Ansible, người dùng thực hiện lệnh git clone để lấy mã nguồn từ kho GitHub.
- Các Playbook sau đó được kiểm tra và chuẩn bị để thực thi.

3.4.4. Chạy Ansible Playbook trên Ansible Control Machine

- Người dùng chạy Playbook trên máy điều khiển Ansible bằng lệnh ansible-playbook.
- Quá trình này thực hiện các hành động sau:
 - Kết nối tới các VM trong cụm thông qua SSH.
 - Triển khai RKE2 trên từng máy ảo.

3.4.5. Deploy RKE2 lên các Master và Worker Node

- Master Nodes:
 - Triển khai các thành phần quan trọng như kube-apiserver, etcd, kube-scheduler và kube-controller-manager.
 - Thiết lập chế độ High Availability (HA) để đảm bảo tính chịu lỗi và khả năng mở rộng.
- Worker Nodes:
 - Cài đặt RKE2 agent để giao tiếp với các master nodes.

- Cấu hình các worker node để thực thi các workloads và đảm bảo khả năng mở rộng của cụm.

3.5. Luồng hoạt động của các ứng dụng trong Cluster

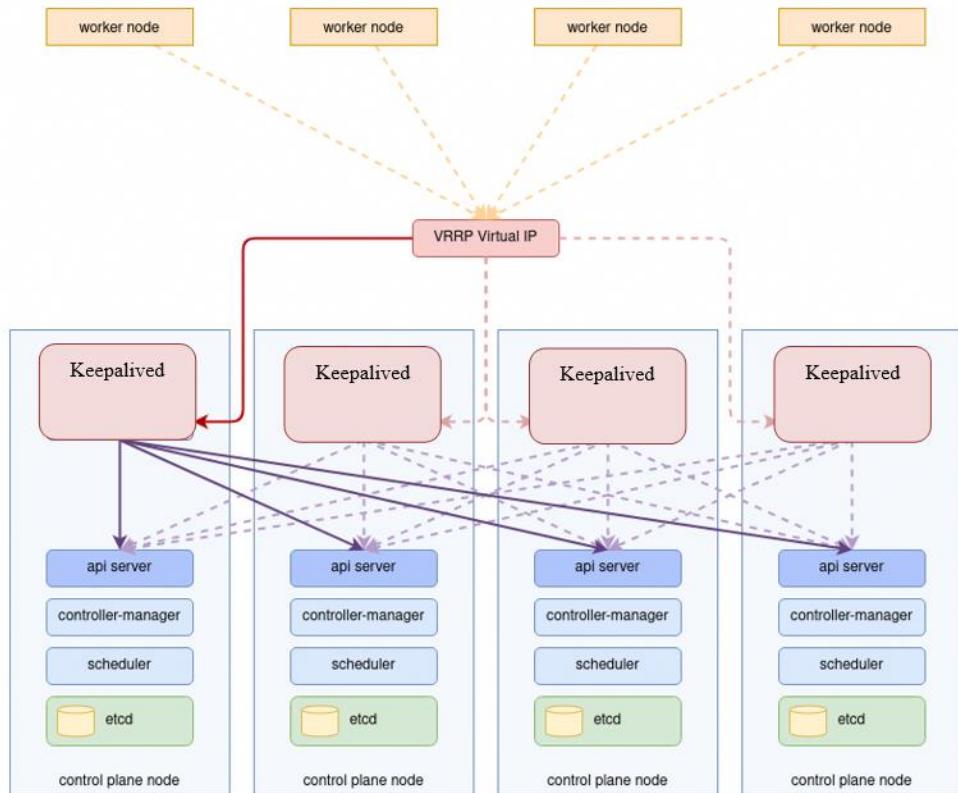
3.5.1. RKE2 HA

3.5.1.1. Keepalived và VRRP VIP:

- Keepalived sử dụng giao thức VRRP để tạo một địa chỉ IP ảo (VIP).
- VIP đóng vai trò là điểm vào cố định để worker nodes và các client (kubectl) giao tiếp với control plane.

3.5.1.2. Luồng giao tiếp:

- Worker Nodes:
 - Gửi yêu cầu đăng ký đến VIP tại cổng 9345.
 - Sau khi đăng ký, worker nodes giao tiếp với Kubernetes API Server thông qua VIP tại cổng 6443.
- Control Plane Nodes:
 - Chạy các thành phần như API Server, Controller Manager, Scheduler, và etcd (stacked).
 - Các control plane nodes giao tiếp trực tiếp với nhau để duy trì trạng thái của cluster và quorum trong etcd.
- Failover (Chuyển đổi VIP):
 - Nếu control plane node đang giữ VIP gặp sự cố, Keepalived sẽ tự động chuyển VIP sang một control plane node khác khỏe mạnh.
 - Worker nodes và client không bị gián đoạn, vì kết nối luôn đi qua VIP.



Hình 3.4: RKE2 HA Topology với Keepalived

3.5.2. Rancher

3.5.2.1. Xác thực người dùng

- Người dùng gửi yêu cầu (như kiểm tra Pod) qua giao diện Rancher hoặc kubectl.
- Rancher dùng Authentication Proxy để xác thực người dùng thông qua các dịch vụ như Active Directory hoặc GitHub.
- Sau đó, Rancher chuyển tiếp yêu cầu này đến API Server của cluster.

3.5.2.2. Cluster Controller và Cluster Agent

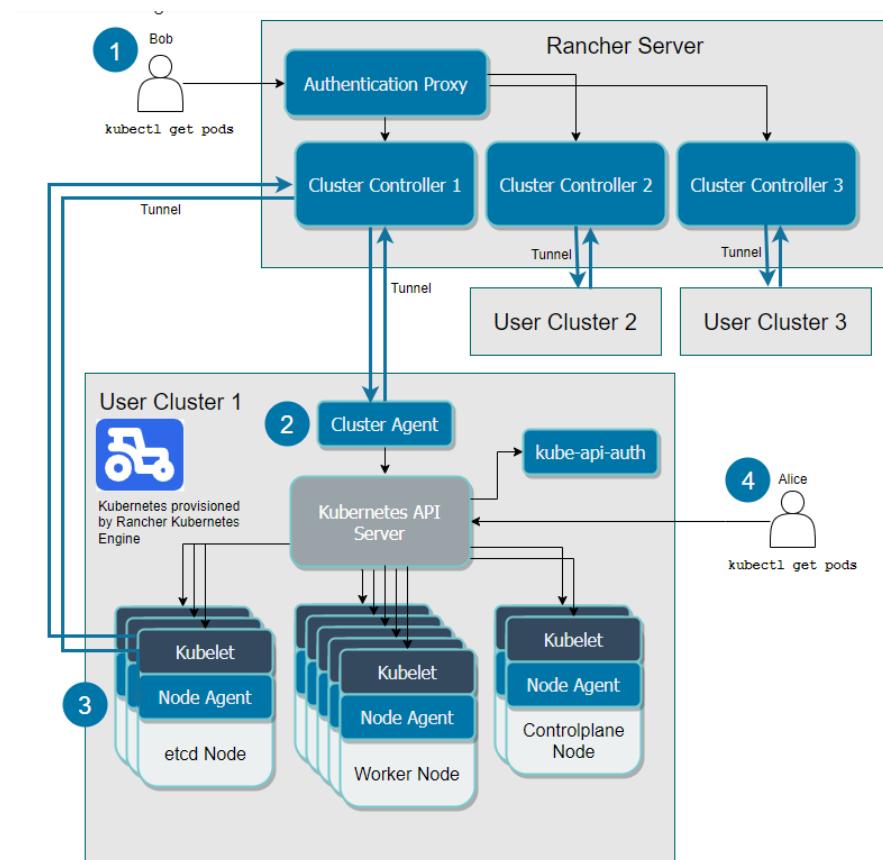
- Cluster Controller: Chạy trong Rancher server, quản lý trạng thái cluster, triển khai tài nguyên, và kiểm soát truy cập (RBAC).
- Cluster Agent: Chạy trong cluster RKE2, nhận lệnh từ Rancher, triển khai Pod, và báo cáo trạng thái cluster.

3.5.2.3. Node Agent

- Chạy trên mỗi node trong cluster để hỗ trợ quản lý node (nâng cấp Kubernetes, khôi phục snapshot, v.v.).
- Tạo kênh dự phòng nếu Cluster Agent không hoạt động.

3.5.2.4. Kết nối trực tiếp (ACE)

- Authorized Cluster Endpoint (ACE): Cho phép kết nối trực tiếp tới API Server của cluster mà không qua Rancher.
- Hữu ích khi Rancher bị lỗi hoặc để giảm độ trễ trong kết nối.



Hình 3.5: Sơ đồ minh họa cách Rancher hoạt động

3.5.3. Longhorn

Longhorn là một hệ thống lưu trữ phân tán dành cho Kubernetes, hoạt động dựa trên các dịch vụ container hóa với các thành phần chính sau:

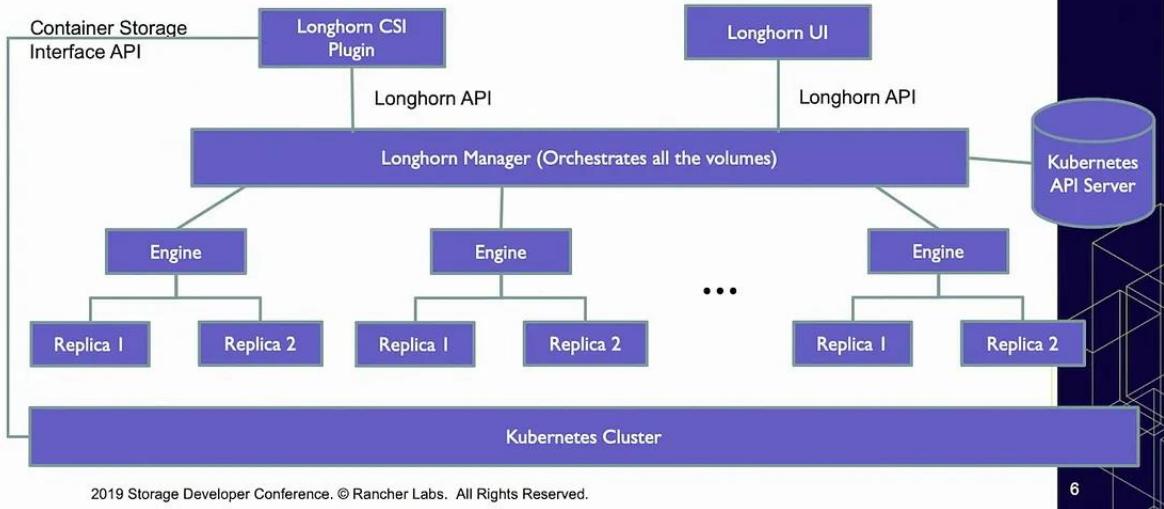
- Longhorn Manager: Là "bộ não" của hệ thống, quản lý việc tạo và giám sát các ổ đĩa (volume), đảm bảo số lượng bản sao (replica) cần thiết và tình trạng sức khỏe của chúng.
- Longhorn Engine: Đảm nhận vai trò truyền và nhận dữ liệu, quản lý các bản sao của từng volume để đảm bảo dữ liệu luôn đồng nhất.
- Replica Pods: Mỗi volume được sao lưu bởi nhiều bản sao (replica) trên các node khác nhau. Nếu một node bị lỗi, các bản sao còn lại đảm bảo dữ liệu không bị mất.
- Snapshots và Backups: Hỗ trợ chụp nhanh dữ liệu (snapshot) và sao lưu (backup) theo thời gian thực, lưu trữ trên cluster hoặc trên các dịch vụ như S3.
- Tách biệt Control Plane và Data Plane: Longhorn tách biệt phần quản lý (control plane) và phần xử lý dữ liệu (data plane), đảm bảo hiệu suất và bảo mật cao.

Khi có yêu cầu lưu trữ, Longhorn sẽ:

- Quản lý: Longhorn Manager phân bổ volume, lên lịch và giám sát bản sao.
- Lưu trữ: Longhorn Engine nhận dữ liệu, đồng bộ hóa các bản sao trên nhiều node.
- Phục hồi: Nếu có lỗi xảy ra, Longhorn Manager tự động tạo lại bản sao ở các node khác.

Longhorn Architecture - Manager

SDC¹⁹



Hình 3.6: Kiến trúc của Longhorn

3.5.4. Docker Registry

Docker Registry là nơi lưu trữ và phân phối các hình ảnh Docker. Dưới đây là cách hoạt động của nó một cách đơn giản:

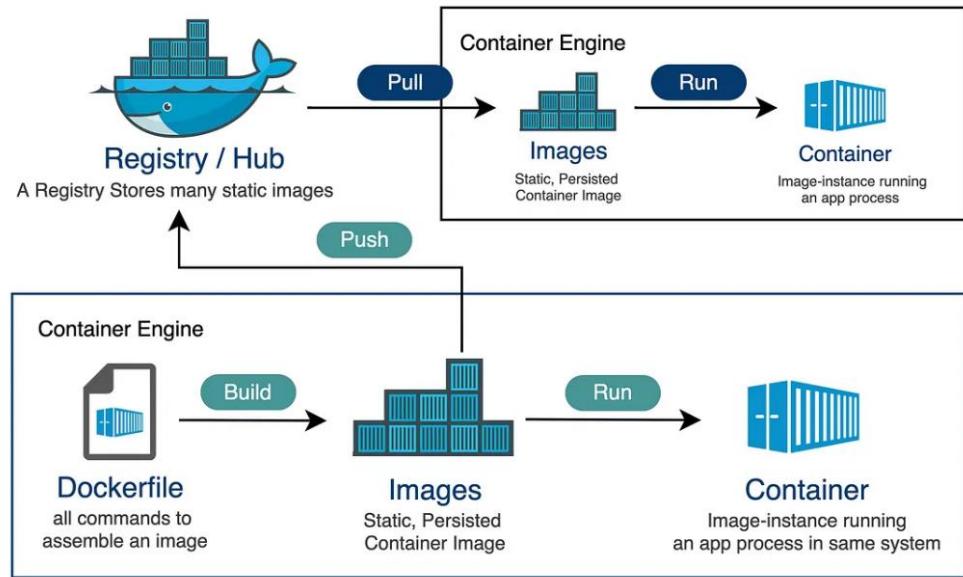
3.5.4.1. Lưu trữ và Phân phối

- Registry: Tưởng tượng như một thư viện, nơi chứa nhiều hình ảnh Docker (như sách).
- Repository: Mỗi repository là một bộ sưu tập các hình ảnh cùng tên nhưng có các phiên bản khác nhau (như các phiên bản khác nhau của một cuốn sách).

3.5.4.2. Quá trình Pull và Push

- Pull: Khi bạn cần một hình ảnh, Docker gửi yêu cầu đến registry để tải hình ảnh về. Quy trình này bao gồm:
 - Yêu cầu manifest (thông tin về các lớp của hình ảnh).
 - Tải xuống các lớp hình ảnh không có sẵn trên máy của bạn.
- Push: Khi bạn muốn lưu trữ một hình ảnh mới, Docker sẽ:
 - Tải lên từng lớp hình ảnh không có trong registry.

- Gửi manifest để lưu trữ thông tin về hình ảnh.



Hình 3.7: Luồng hoạt động của Docker Registry

3.5.5. Metallb

3.5.5.1. Cáp phát địa chỉ IP (IP Pool)

- MetallLB được cấu hình ở chế độ Layer 2 hoặc BGP (trong môi trường của bạn, khả năng cao là Layer 2).
- Một IP Pool được định nghĩa trong MetalLB config map (ví dụ: dải IP từ MetalLB có thể là 192.168.1.100-192.168.1.200).
- Khi một dịch vụ trong cluster yêu cầu LoadBalancer (ví dụ, Rancher, JupyterHub), MetalLB chọn một IP từ IP Pool và gán cho dịch vụ đó.

3.5.5.2. Giao tiếp dịch vụ

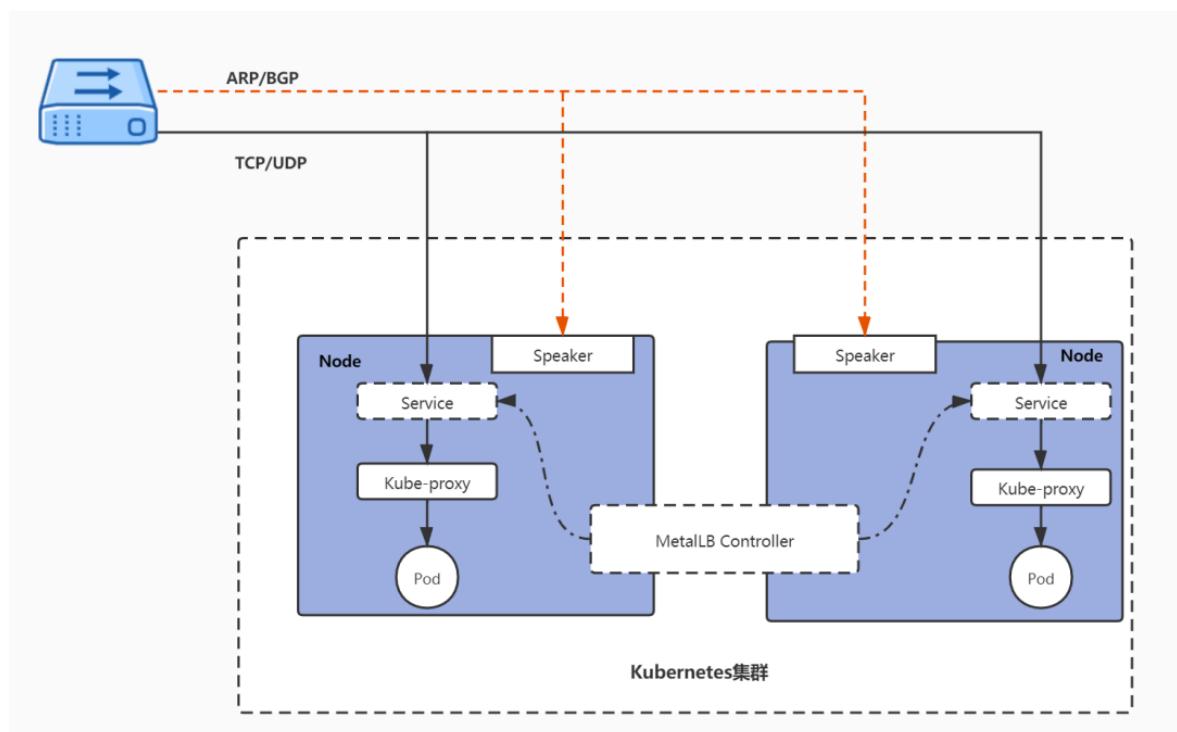
- Khi một ứng dụng (ví dụ, Rancher hoặc JupyterHub) yêu cầu LoadBalancer, MetalLB gán một IP từ IP Pool.
- Các worker nodes nhận biết IP này thông qua ARP hoặc thông báo BGP (trong chế độ tương ứng), đảm bảo lưu lượng được phân phối đến đúng node chạy pod của dịch vụ.

3.5.5.3. Chuyển tiếp lưu lượng

Khi một yêu cầu từ bên ngoài đến IP LoadBalancer:

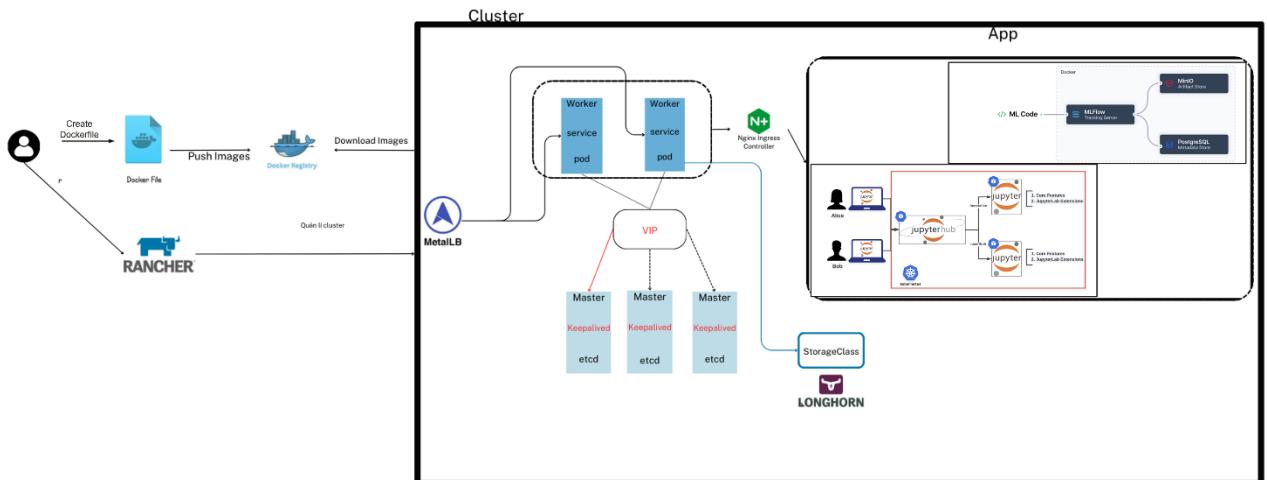
- MetalLB định tuyến yêu cầu đến node tương ứng chạy dịch vụ.
- Kube-Proxy và CNI (Canal trong trường hợp của bạn) chuyển tiếp lưu lượng đến đúng pod.

Nếu dịch vụ có nhiều pod, kube-proxy sử dụng kỹ thuật round-robin hoặc IPVS để phân phối lưu lượng.



Hình 3.8: Luồng hoạt động của Metallb

3.5.6. Tổng quan luồng hoạt động của hệ thống



Hình 3.9: Luồng hoạt động của hệ thống

3.5.6.1. Tạo và quản lý container qua Rancher:

- Rancher đóng vai trò là giao diện quản lý Kubernetes giúp tạo các tài nguyên như Deployment, Service, và Pods.
- Cho phép thiết lập cấu hình cluster, thông số hoạt động, và quản lý quyền truy cập dựa trên vai trò (RBAC).

3.5.6.2. Docker Registry:

- Người dùng tạo hình ảnh Docker từ các tệp Dockerfile và đẩy lên Docker Registry.
- Docker Registry lưu trữ hình ảnh Docker cục bộ, đảm bảo an toàn và tốc độ tải xuống nhanh chóng khi cần triển khai.
- Khi cần, các worker nodes sẽ tải hình ảnh từ Docker Registry để triển khai các ứng dụng.

3.5.6.3. Cluster RKE2 với HA (High Availability):

Master Nodes: Sử dụng RKE2 với Keepalived để đảm bảo tính sẵn sàng cao cho master nodes.

- Các master nodes chứa etcd để lưu trữ trạng thái của cluster.

- Keepalived cung cấp một IP ảo (VIP) giúp các worker nodes luôn có thể kết nối tới một master node đang hoạt động.

Worker Nodes: Các worker nodes chứa các Pods triển khai ứng dụng.

- MetalLB hoạt động như Load Balancer để phân phối lưu lượng đến các service trong cluster.
- Nginx Ingress Controller quản lý lưu lượng HTTP/HTTPS đến các ứng dụng.

3.5.6.4. Lưu trữ với Longhorn:

Longhorn cung cấp giải pháp lưu trữ phân tán cho cluster, đảm bảo dữ liệu của Pods được lưu trữ bền vững và có thể truy cập được ngay cả khi node gặp sự cố.

3.5.6.5. Ứng dụng và dịch vụ triển khai trên cluster:

JupyterHub:

- Người dùng (ví dụ: Alex, Bob) sử dụng JupyterHub để tạo các môi trường làm việc tách biệt thông qua container.
- Mỗi môi trường Jupyter Notebook được hỗ trợ bởi tài nguyên cluster (CPU, RAM, lưu trữ).

MLflow:

- MLflow được sử dụng để quản lý vòng đời các mô hình học máy, bao gồm theo dõi, lưu trữ mô hình, và triển khai.
- MinIO đóng vai trò như một object storage để lưu trữ dữ liệu huấn luyện và mô hình học máy.
- PostgreSQL được sử dụng làm cơ sở dữ liệu backend cho MLflow.

3.5.6.6. Giám sát và cảnh báo:

- Kube-Prometheus Stack (bao gồm Prometheus, Grafana, và Alertmanager) được triển khai để giám sát cluster và đưa ra các cảnh báo khi phát hiện bất thường.
- Người dùng có thể theo dõi các thông số như CPU, RAM, I/O và dung lượng lưu trữ thông qua giao diện của Grafana.

Chương 4. Hiện thực hệ thống

4.1. Công cụ và môi trường

- Hệ điều hành: Ubuntu 22.04/ Windows
- Môi trường: Vmware, Cloud
- Công cụ tự động hóa cài đặt: Ansible, bash script
- Công cụ chính: KE2, Helm, Prometheus, Grafana, Longhorn, MinIO, PostgreSQL, MLflow, JupyterHub.

4.2. Triển khai Kubernetes

4.2.1. Triển khai RKE2 cluster bằng Ansible

- Ansible là công cụ tự động hóa mạnh mẽ giúp đơn giản hóa việc triển khai RKE2 trên nhiều node (master và worker). Sử dụng Ansible sẽ giảm thiểu lỗi thủ công, tăng tốc quá trình thiết lập cluster Kubernetes.
- Nhóm đã tạo một github tích hợp Ansible Playbook của Labyrinth Labs (lablabs) với Script Ansible tự viết để chuẩn bị cho từng node và sử dụng bash script để tải các công cụ khác .

4.2.1.1. Yêu cầu trước khi thực hiện

- Cài đặt Ansible trên máy điều khiển (control machine).
- Truy cập SSH được cấu hình trên tất cả các node (master và worker) với cùng một tài khoản người dùng:

```
sudo apt install -y openssh-server  
sudo systemctl enable ssh  
sudo systemctl start ssh
```

- Mỗi node phải có ít nhất 4G RAM, 2 CPU, 80G ổ cứng để quá trình cài đặt diễn ra thuận tiện nhất.
- Đổi hostname cho máy ansible:

```
sudo hostnamectl set-hostname ansible  
exec bash
```

4.2.1.2. Triển khai Script

III. Tùy chọn, ghi đè các biến qua dòng lệnh:

1. Chế độ Normal (1 Master và 1 hoặc nhiều Worker):

```
sudo USERNAME=ubuntu \  
ANSIBLE_HOST_IP=192.168.198.129 \  
MASTER_IPS="192.168.198.141" \  
WORKER_IPS="192.168.198.132 192.168.198.133" \  
RKE2_MODE=normal \  
RKE2_CNI=canal \  
RKE2_TOKEN="yourSecureToken123" \  
../ansible.sh
```

2. Chế độ High Availability (HA) (Nhiều Master và 1 hoặc nhiều Worker):

```
sudo USERNAME=ubuntu \  
ANSIBLE_HOST_IP=192.168.198.129 \  
MASTER_IPS="192.168.198.141 192.168.198.142 192.168.198.143" \  
WORKER_IPS="192.168.198.132" \  
API_IP=192.168.198.100 \  
RKE2_MODE=ha \  
RKE2_CNI=canal \  
RKE2_TOKEN="yourSecureToken123" \  
RKE2_LOADBALANCER_RANGE="192.168.198.10-192.168.198.110" \  
USE_KUBEVIP=false \  
../ansible.sh
```

Hình 4.1: Câu lệnh để triển khai RKE2 cluster

Giải thích các biến môi trường:

- **USERNAME** (Mặc định: ubuntu): Tên người dùng SSH để đăng nhập vào các node master và worker. Ghi đè biến này nếu tên người dùng của bạn khác với mặc định ubuntu
- **ANSIBLE_HOST_IP** (Mặc định: 192.168.198.129): Địa chỉ IP của máy chủ Ansible. Đây là địa chỉ của máy chạy Ansible và thực hiện các kết nối SSH đến các node khác
- **MASTER_IPS** (Mặc định: 192.168.198.141): Danh sách các địa chỉ IP cho các node master trong cluster Kubernetes của bạn. Đảm bảo danh sách này bao gồm tất cả các địa chỉ IP của các node master nếu bạn đang cài đặt một cluster có nhiều master.

- WORKER_IPS (Mặc định: 192.168.198.132): Danh sách các địa chỉ IP cho các node worker trong cluster Kubernetes của bạn. Bao gồm tất cả các địa chỉ IP của các node worker mà bạn muốn thêm vào cluster.
- API_IP (Cần thiết cho chế độ HA): Địa chỉ IP được sử dụng cho bộ cân bằng tải API Kubernetes trong một cấu hình High Availability (HA). Để trống hoặc không cung cấp trong chế độ bình thường, nhưng cần thiết trong chế độ HA.
- RKE2_MODE (Mặc định: normal): Chế độ cài đặt cho RKE2.

Có thể chọn:

normal: Cài đặt với một node master và một hoặc nhiều node worker.

ha: Cài đặt High Availability (HA) với nhiều node master và một hoặc nhiều node worker.

- RKE2_CNI (Mặc định: canal): Plugin Container Network Interface (CNI) sử dụng cho mạng trong cluster RKE2 của bạn. canal là mặc định, nhưng bạn có thể chọn các CNI khác hỗ trợ bởi RKE2, ví dụ flannel hoặc calico.
- RKE2_TOKEN : Token RKE2 được sử dụng để xác thực giữa các node trong cluster. Thay thế token này bằng token bảo mật của bạn để đảm bảo việc giao tiếp giữa các node.
- RKE2_LOADBALANCER_RANGE (Yêu cầu trong chế độ HA): Dải IP cho load balancer trong chế độ HA. Ví dụ: 192.168.198.200-192.168.198.220.
- USE_KUBEVIP=false: Đặt biến môi trường này là true nếu bạn muốn sử dụng KubeVIP trong chế độ HA (High Availability). Nếu bạn đặt nó là false, Keepalived sẽ được sử dụng thay thế.

4.2.1.3. Giải thích Script

a) ansible.sh:

Script được viết bằng Bash để triển khai và cấu hình RKE2 (Rancher Kubernetes Engine 2) trên các máy chủ sử dụng Ansible. Dưới đây là giải thích ngắn gọn từng phần:

- Thiết lập giá trị mặc định:
 - Sử dụng các biến môi trường với giá trị mặc định như USERNAME, RKE2_VERSION, RKE2_MODE, v.v., để cấu hình cluster.
 - Chế độ hỗ trợ: normal (1 master) hoặc ha (High Availability với nhiều master).

```
# Default values for variables
: "${USERNAME:=ubuntu}"                                # SSH username
: "${ANSIBLE_HOST_IP:=192.168.198.129}"               # Control machine IP
: "${RKE2_VERSION:=v1.30.6+rke2r1}"                   # Default RKE2 version
: "${RKE2_MODE:=normal}"                               # Default mode (normal or ha)
: "${RKE2_TOKEN:=yourSecureToken123}"                 # Default RKE2 token
: "${API_IP:="}"                                     # Default API IP (empty for normal mode)
: "${RKE2_CNI:=canal}"                               # Default CNI (Container Network Interface)
: "${RKE2_LOADBALANCER_RANGE:="}"                    # Default load balancer IP range
: "${USE_KUBEVIP:=false}"                            # Default to keepalived, set to true for kubevip
: "${RKE2_HA_MODE_KUBEVIP:=false}"                  # Default HA mode kubevip (false)
: "${RKE2_HA_MODE_KEEPALIVED:=true}"                # Default HA mode keepalived (true)

# Arrays of IPs for masters and workers
MASTER_IPS=(${MASTER_IPS:-192.168.198.141})
WORKER_IPS=(${WORKER_IPS:-192.168.198.132})
```

Hình 4.2: Giá trị mặc định

- Chuẩn bị máy Ansible
 - Cài đặt các gói cần thiết như Ansible, SSH.
 - Cài đặt role Ansible từ lablabs.rke2.

```
# Prepare Ansible machine
sudo apt update
sudo apt install ansible -y
sudo apt install curl -y
ansible-galaxy install lablabs.rke2 --force
```

Hình 4.3: Câu lệnh cài đặt các gói cần thiết

- SSH và sao chép khóa: Tạo khóa SSH (nếu chưa có) và sao chép chúng đến các máy master và worker để thiết lập kết nối không cần mật khẩu.

```

if ! dpkg -l | grep -q openssh-server; then
    echo "OpenSSH Server is not installed. Installing..."
    sudo apt install -y openssh-server
else
    echo "OpenSSH Server is already installed."
fi

# Enable and start SSH service if not already running
if ! systemctl is-active --quiet ssh; then
    echo "Starting SSH service..."
    sudo systemctl enable ssh
    sudo systemctl start ssh
else
    echo "SSH service is already running."
fi

# Generate SSH keys if they don't already exist
if [ ! -f "$HOME/.ssh/id_rsa" ]; then
    ssh-keygen -t rsa -N "" -f "$HOME/.ssh/id_rsa"
fi

# Copy SSH keys to master and worker nodes
for MASTER_IP in "${MASTER_IPS[@]}"; do
    ssh-copy-id "$USERNAME@$MASTER_IP"
done

for WORKER_IP in "${WORKER_IPS[@]}"; do
    ssh-copy-id "$USERNAME@$WORKER_IP"
done

```

Hình 4.4: Cài đặt SSH và sao chép khóa ssh

- Cập nhật file /etc/hosts Thêm các địa chỉ IP của máy chủ điều khiển (control machine), master, và worker vào file /etc/hosts để dễ dàng nhận diện qua hostname.

```

# Function to add IPv4 entries to /etc/hosts
add_to_hosts() {
    local ip=$1
    local hostname=$2

    # Remove any existing entry with the same hostname, regardless of IP
    sudo sed -i "/[:space:]\+$hostname/d" /etc/hosts

    # Add the new entry, ensuring it's added before the IPv6 section
    sudo sed -i "/^# The following lines are desirable for IPv6 capable hosts/i\\\$ip \$hostname" /etc/hosts
}

# Add control machine and node IPs to /etc/hosts
add_to_hosts "$ANSIBLE_HOST_IP" "ansible"

# Update master nodes in /etc/hosts
for i in "${!MASTER_IPS[@]}"; do
    MASTER_IP=${MASTER_IPS[i]}
    add_to_hosts "$MASTER_IP" "master$((i+1))"
done

# Update worker nodes in /etc/hosts
for i in "${!WORKER_IPS[@]}"; do
    WORKER_IP=${WORKER_IPS[i]}
    add_to_hosts "$WORKER_IP" "worker$((i+1))"
done

```

Hình 4.5: Cập nhật file /etc/hosts trên máy Ansible

- Tạo file hosts cho Ansible: Tạo file inventory hosts để liệt kê các master và worker, đồng thời định nghĩa nhóm k8s_cluster cho Ansible.

```
# Create an Ansible hosts file with master and worker configurations
cat > hosts <<EOF
[masters]
EOF

for i in "${!MASTER_IPS[@]}"; do
    MASTER_IP=${MASTER_IPS[i]}
    echo "master$((i+1)) ansible_host=$MASTER_IP ansible_user=$USERNAME rke2_type=server" >> hosts
done

cat >> hosts <<EOF

[workers]
EOF

for i in "${!WORKER_IPS[@]}"; do
    WORKER_IP=${WORKER_IPS[i]}
    echo "worker$((i+1)) ansible_host=$WORKER_IP ansible_user=$USERNAME rke2_type=agent" >> hosts
done

cat >> hosts <<EOF

[k8s_cluster:children]
masters
workers
EOF
```

Hình 4.6: Tạo file hosts cho Ansible Script

- Lấy phiên bản RKE2 mới nhất: Gửi yêu cầu đến API GitHub để lấy phiên bản RKE2 mới nhất.

```
# Function to get the latest RKE2 version from GitHub
get_latest_rke2_version() {
    curl -s "https://api.github.com/repos/rancher/rke2/releases/latest" | \
    grep '"tag_name":' | \
    sed -E 's/.+"(["]+)"./\1/' \
}

# Set the RKE2_VERSION dynamically
RKE2_VERSION=$(get_latest_rke2_version)
if [[ -z "$RKE2_VERSION" ]]; then
    echo "Failed to fetch the latest RKE2 version. Using default version: v1.30.6+rke2r1"
    RKE2_VERSION="v1.30.6+rke2r1"
fi
```

Hình 4.7: Lấy phiên bản RKE2 mới nhất

- Triển khai cluster RKE2. Chạy playbook Ansible theo chế độ:
 - Normal mode: 1 master, nhiều worker.

```
# Prompt for the 'become' password (sudo access)
echo "Please enter the become password:"
read -s BECOME_PASS

# Set the ANSIBLE_BECOME_PASS environment variable for Ansible
export ANSIBLE_BECOME_PASS=$BECOME_PASS

# Deploy RKE2 based on the mode (normal or ha)
if [[ $RKE2_MODE == "normal" && ${#MASTER_IPS[@]} -eq 1 && ${#WORKER_IPS[@]} -ge 1 ]]; then
    ansible-playbook -i hosts tasks/prepare_vm.yaml

    # Normal mode: one master and one or more workers
    ansible-playbook -i hosts tasks/deploy_rke2.yaml \
        --extra-vars "rke2_cni=$RKE2_CNI rke2_version=$RKE2_VERSION rke2_token=$RKE2_TOKEN"

    # Run the post_install.yaml playbook for additional setup on master nodes
    ansible-playbook -i hosts tasks/post_install.yaml --user=root

```

Hình 4.8: Triển khai RKE2 normal mode

- HA mode: Nhiều master với Keepalived hoặc Kube-VIP. Cần thiết lập thêm API_IP và RKE2_LOADBALANCER_RANGE cho chế độ HA.

```
elif [[ $RKE2_MODE == "ha" && ${#MASTER_IPS[@]} -gt 1 && ${#WORKER_IPS[@]} -ge 1 ]]; then
    # HA mode: multiple masters and one or more workers
    if [[ -z $API_IP || -z $RKE2_LOADBALANCER_RANGE ]]; then
        echo "In HA mode, API_IP and RKE2_LOADBALANCER_RANGE must be set. Exiting."
        exit 1
    fi

    # Set kubevip or keepalived based on user input
    if [[ "$USE_KUBEVIP" == "true" ]]; then
        HA_MODE_KUBEVIP=true
        HA_MODE_KEEPALIVED=false
    else
        HA_MODE_KUBEVIP=false
        HA_MODE_KEEPALIVED=true
    fi

    # Run the Ansible playbooks for setup and deployment
    ansible-playbook -i hosts tasks/prepare_vm.yaml --extra-vars "api_ip=$API_IP"
    ansible-playbook -i hosts tasks/deploy_rke2_ha.yaml \
        --extra-vars "rke2_cni=$RKE2_CNI rke2_version=$RKE2_VERSION rke2_token=$RKE2_TOKEN rke2_api_ip=$API_IP"

    # Run the post_install.yaml playbook for additional setup on master nodes
    ansible-playbook -i hosts tasks/post_install.yaml --user=root

else
    echo "Invalid configuration: Please check RKE2_MODE, MASTER_IPS, and WORKER_IPS."
    exit 1
fi

```

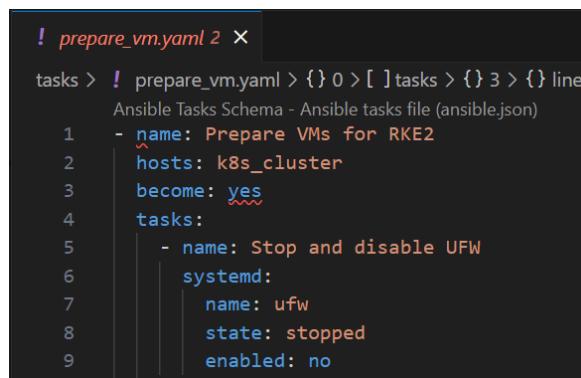
Hình 4.9: Triển khai RKE2 HA mode

- b) prepare_vm.yaml

Chuẩn bị các máy chủ (Master và Worker) để triển khai RKE2 Cluster.

Các bước thực hiện:

- Tắt và vô hiệu hóa Firewall (UFW): Dừng và ngăn khởi động dịch vụ UFW trên các máy chủ.



```
! prepare_vm.yaml 2 ×
tasks > ! prepare_vm.yaml > {} 0 > [ ] tasks > {} 3 > {} line1
Ansible Tasks Schema - Ansible tasks file (ansible.json)
1   - name: Prepare VMs for RKE2
2     hosts: k8s_cluster
3     become: yes
4     tasks:
5       - name: Stop and disable UFW
6         systemd:
7           name: ufw
8           state: stopped
9           enabled: no
```

Hình 4.10: Vô hiệu hóa ufw

- Cấu hình IP Forwarding:
 - Tạo file /etc/sysctl.d/90-rke2.conf.
 - Bật chuyển tiếp IP cho IPv4 và IPv6 bằng cách thêm:

net.ipv4.ip_forward=1

net.ipv6.conf.all.forwarding=1

- Áp dụng thay đổi bằng sysctl --system.

```

- name: Create sysctl configuration file
  file:
    path: /etc/sysctl.d/90-rke2.conf
    state: touch

- name: Enable IP forwarding for IPv4
  lineinfile:
    path: /etc/sysctl.d/90-rke2.conf
    line: 'net.ipv4.ip_forward=1'
    state: present

- name: Enable IP forwarding for IPv6
  lineinfile:
    path: /etc/sysctl.d/90-rke2.conf
    line: 'net.ipv6.conf.all.forwarding=1'
    state: present

- name: Apply sysctl settings
  command: sysctl --system
  changed_when: false

```

Hình 4.11: Cấu hình IP Forwarding

- Tắt và vô hiệu hóa Swap: Chạy lệnh swapoff -a. Chính sửa file /etc/fstab để bình luận các dòng cấu hình liên quan đến Swap.

```

- name: Disable swap
  command: swapoff -a
  ignore_errors: yes
  changed_when: false

- name: Comment out swap in fstab
  lineinfile:
    path: /etc/fstab
    regexp: '^([^\#].*swap.*)'
    line: '# \1'
    state: present

```

Hình 4.12: Tắt và vô hiệu hóa Swap

- Cập nhật và nâng cấp hệ thống:
 - Cập nhật danh sách gói.
 - Cài đặt và nâng cấp gói nfs-common và các gói hệ thống khác.
 - Tự động gỡ bỏ các gói không cần thiết.

```

- name: Update package lists
  apt:
    update_cache: yes

- name: Install nfs-common
  apt:
    name: nfs-common
    state: present

- name: Upgrade packages
  apt:
    upgrade: dist
    state: latest

- name: Autoremove unnecessary packages
  apt:
    autoremove: yes

```

Hình 4.13: Cập nhật và nâng cấp hệ thống

- Cập nhật file /etc/hosts:
 - HA Mode: Thêm API IP với hostname lb.
 - IPv4 Section: Thêm các địa chỉ IP và hostname từ file inventory.
 - Thiết lập hostname: Cập nhật hostname theo inventory_hostname (cho cả Master và Worker).

```

- name: Add API IP with hostname lb to /etc/hosts (only in HA mode)
  lineinfile:
    path: /etc/hosts
    line: "{{ api_ip }} lb"
    create: yes
    when: api_ip is defined

- name: Update /etc/hosts file with Ansible inventory in IPv4 section
  blockinfile:
    path: /etc/hosts
    marker: "# Added by Ansible - IPv4 Section"
    block: |
      {% for item in groups['all'] %}
        {{ hostvars[item].ansible_host }} {{ item }}
      {% endfor %}
    state: present
    when: ansible_host is defined

- name: Set hostname based on inventory_hostname
  hostname:
    name: "{{ inventory_hostname }}"
  when: inventory_hostname in groups['masters'] + groups['workers']

```

Hình 4.14: Cập nhật file /etc/hosts và hostname

- Cài đặt Open-iSCSI (chỉ Worker): Đảm bảo các node Worker cài đặt gói open-iscsi.

```
- name: Install open-iscsi (only on workers)
  apt:
    name: open-iscsi
    state: present
    when: inventory_hostname in groups['workers']
```

Hình 4.15: Cài đặt Open-iSCSI

- Khởi động lại máy: Reboot để áp dụng toàn bộ thay đổi.

```
- name: Reboot the server to apply changes
  reboot:
    msg: "Rebooting to apply changes"
    connect_timeout: 5
    when: inventory_hostname in groups['masters'] + groups['workers']
```

Hình 4.16: Khởi động lại

c) deploy_rke2.yaml

- Mục tiêu: Triển khai RKE2 trên các node trong cluster.
- Hosts: Chạy trên nhóm máy k8s_cluster.
- Quyền truy cập: Sử dụng quyền become: yes để chạy các lệnh với quyền sudo.
- Biến sử dụng:
 - rke2_version: Phiên bản RKE2 cần cài đặt.
 - rke2_cni: Plugin mạng (CNI) sử dụng, ví dụ: Canal hoặc Cilium.
 - rke2_token: Token dùng để kết nối các node với cluster.
 - rke2_server_node_taints: Cấu hình taints để chỉ định các node server chỉ chạy các dịch vụ cần thiết (ví dụ: CriticalAddonsOnly=true:NoExecute).
- Vai trò: Sử dụng role lablabs.rke2 để thực hiện việc cài đặt và cấu hình RKE2.

```

! deploy_rke2.yaml 2 ×

tasks > ! deploy_rke2.yaml > {} 0 > [ ]roles > {} 0 > role
Ansible Tasks Schema - Ansible tasks file (ansible.json)
1   - name: Deploy RKE2
2     hosts: k8s_cluster
3     become: yes
4     vars:
5       rke2_version: "{{ rke2_version }}"
6       rke2_cni: ["{{ rke2_cni }}"] # or [cilium]
7       rke2_token: "{{ rke2_token }}"
8       rke2_server_node_taints:
9         - 'CriticalAddonsOnly=true:NoExecute'
10      roles:
11        - role: lablabs.rke2

```

Hình 4.17: File deploy_rke2.yaml

d) deploy_rke2_ha.yaml

- Mục tiêu: Triển khai cluster RKE2 với cấu hình High Availability (HA).
- Hosts: Áp dụng trên nhóm máy k8s_cluster.
- Quyền truy cập: Sử dụng quyền become: yes để thực thi lệnh với quyền root.
- Các biến cấu hình:
 - RKE2 phiên bản: rke2_version – Xác định phiên bản RKE2 cần cài đặt.
 - Plugin mạng (CNI): rke2_cni – Sử dụng plugin mạng (ví dụ: Canal hoặc Cilium).
 - Token cluster: rke2_token – Token để các node kết nối với cluster.
 - Chế độ HA: rke2_ha_mode: Bật chế độ HA.
 - Chọn HA Mode:
 - + rke2_ha_mode_kubevip: Sử dụng KubeVIP để quản lý IP API server.
 - + rke2_ha_mode_keepalived: Sử dụng Keepalived để đảm bảo IP ảo (VIP).
- API server IP: rke2_api_ip – Địa chỉ IP của API server.
- Tải file kubeconfig: rke2_download_kubeconf: Cho phép tải kubeconfig sau khi cài đặt.

- SAN bổ sung: rke2_additional_sans: Thêm các địa chỉ IP/hostnames vào SAN để truy cập API server (bao gồm IP của master nodes và API IP).
- Dải IP cho Load Balancer: rke2_loadbalancer_ip_range: Dải IP để MetalLB hoạt động.
- Taints cho server node: CriticalAddonsOnly=true:NoExecute: Chỉ cho phép chạy các Add-ons quan trọng trên các server nodes.

```
! deploy_rke2_ha.yaml 2 ×
tasks > ! deploy_rke2_ha.yaml > {} 0 > [ ]roles > {} 0
Ansible Tasks Schema - Ansible tasks file (ansible.json)
1   - name: Deploy RKE2
2     hosts: k8s_cluster
3     become: yes
4     vars:
5       rke2_version: "{{ rke2_version }}"
6       rke2_cni: ["{{ rke2_cni }}"] # or [cilium]
7       rke2_token: "{{ rke2_token }}"
8       rke2_ha_mode: true
9       rke2_ha_mode_kubevip: "{{ rke2_ha_mode_kubevip }}"
10      rke2_ha_mode_keepalived: "{{ rke2_ha_mode_keepalived }}"
11      rke2_api_ip: "{{ rke2_api_ip }}"
12      rke2_download_kubeconf: true
13      rke2_additional_sans:
14        | {{ groups['masters'] | map('extract', hostvars, 'ansible_host') | list + [rke2_api_ip] }}
15      rke2_loadbalancer_ip_range:
16        | range-global: "{{ rke2_loadbalancer_range }}"
17      rke2_server_node_taints:
18        | - 'CriticalAddonsOnly=true:NoExecute'
19      roles:
20        - role: lablabs.rke2
```

Hình 4.18: File deploy_rke2_ha.yaml

e) post_install.yaml

Chạy script post_install.sh trên các node master để thực hiện một số cài đặt và cấu hình hậu xử lý cho cluster RKE2.

Các bước thực hiện:

- Đảm bảo script post_install.sh tồn tại: Tạo file /tmp/post_install.sh trên node master với nội dung script.

```
---
- name: Execute post_install.sh on master nodes
  hosts: masters
  become: true
  tasks:
    - name: Ensure the post_install.sh script is present
      copy:
        dest: /tmp/post_install.sh
        content: |
          #!/bin/bash
```

Hình 4.19: Tạo file /tmp/post_install.sh trên node master

- Cập nhật danh sách gói:
 - Chạy apt-get update.
 - Cài đặt các gói cần thiết: python3-pip, docker.io.

```
sudo apt-get update

sudo apt install python3-pip -y

sudo apt install docker.io -y
```

Hình 4.20: Cập nhật và tải gói cần thiết

- Kiểm tra và cài đặt curl (nếu chưa có): Nếu curl chưa được cài đặt, thực hiện cài đặt.

```
# Check and install curl if it's not installed
if ! command -v curl &> /dev/null; then
  echo "Installing curl..."
  sudo apt update && sudo apt install -y curl
else
  echo "curl is installed: $(curl --version | head -n 1)"
fi
```

Hình 4.21: Kiểm tra và cài đặt curl

- Cài đặt Helm (nếu chưa có): Tải và cài đặt Helm phiên bản 3.

```

# Install Helm if it's not installed
if ! command -v helm &> /dev/null; then
    echo "Installing Helm..."
    curl -fsSL https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
else
    echo "Helm is installed: $(helm version --short)"
fi

```

Hình 4.22: Cài đặt Helm

- Tạo cấu hình cho Helm chart: Tạo file /var/lib/rancher/rke2/server/manifests/rke2-ingress-nginx-config.yaml để cấu hình Ingress NGINX với:

use-forwarded-headers: true

enable-ssl-passthrough: true

```

# Write configuration for Helm chart
mkdir -p /var/lib/rancher/rke2/server/manifests
cat <<EOF > /var/lib/rancher/rke2/server/manifests/rke2-ingress-nginx-config.yaml
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-ingress-nginx
  namespace: kube-system
spec:
  valuesContent: |
    controller:
      config:
        use-forwarded-headers: true
      extraArgs:
        enable-ssl-passthrough: true
EOF

```

Hình 4.23: Cấu hình Ingress NGINX

- Tạo symlink cho kubectl: Tìm đường dẫn của kubectl trong /var/lib/rancher/rke2/data/ và tạo symlink tới /usr/local/bin/kubectl.

```

# Symlink kubectl
KUBECTL_PATH=$(find /var/lib/rancher/rke2/data/ -name kubectl 2>/dev/null)
if [[ -n "$KUBECTL_PATH" ]]; then
    echo "Creating symlink for kubectl..."
    sudo ln -sf "$KUBECTL_PATH" /usr/local/bin/kubectl
else
    echo "kubectl not found in /var/lib/rancher/rke2/data/"
    exit 1
fi

```

Hình 4.24: Tạo symlink cho kubectl

- Cấu hình biến môi trường KUBECONFIG: Thêm export KUBECONFIG=/etc/rancher/rke2/rke2.yaml vào file ~/.bashrc và load lại cấu hình.

```
# Set KUBECONFIG environment variable
echo "export KUBECONFIG=/etc/rancher/rke2/rke2.yaml" >> ~/.bashrc
source ~/.bashrc
```

Hình 4.25: Cấu hình biến môi trường KUBECONFIG

- Đặt quyền cho script post_install.sh: Gán quyền thực thi (0755) cho script.
- Chạy script post_install.sh: Thực thi script để áp dụng các cài đặt.
- Xóa script post_install.sh sau khi hoàn tất: Dọn dẹp file /tmp/post_install.sh để tránh lưu trữ file không cần thiết.

```
- name: Set correct permissions on post_install.sh
  file:
    path: /tmp/post_install.sh
    mode: '0755'

- name: Execute post_install.sh
  command: /tmp/post_install.sh

- name: Cleanup post_install.sh
  file:
    path: /tmp/post_install.sh
    state: absent
```

Hình 4.26: Đặt quyền, chạy và xóa script sau khi hoàn tất

4.3. Triển khai các ứng dụng quản lí Cluster

4.3.1. Rancher

Cài đặt hoặc nâng cấp cert-manager và Rancher bằng Helm trên Kubernetes cluster [4].

Chi tiết các bước:

- Cài đặt các biến môi trường:
 - HOSTNAME: Tên miền của Rancher (rancher.mylab.com).
 - BOOTSTRAP_PASSWORD: Mật khẩu mặc định để truy cập Rancher (admin).
- Thêm Helm Repositories:
 - Thêm repo Helm cho Rancher: <https://releases.rancher.com/server-charts/latest>.
 - Thêm repo Helm cho Jetstack (cert-manager): <https://charts.jetstack.io>.
- Cập nhật Helm Repositories: Chạy lệnh helm repo update để đảm bảo có danh sách biểu đồ Helm mới nhất.
- Cài đặt hoặc nâng cấp cert-manager:
 - Sử dụng Helm để triển khai cert-manager với namespace cert-manager.
 - Bao gồm việc áp dụng các CRD cần thiết (crds.enabled=true).

```
$ 2-install_rancher.sh M X
tasks > bash_script > $ 2-install_rancher.sh
1  #!/bin/bash
2
3  # Set environment variables
4  export HOSTNAME="rancher.mylab.com"
5  export BOOTSTRAP_PASSWORD="admin"
6
7  # Add the Rancher and Jetstack Helm repositories
8  echo "Adding Rancher and Jetstack Helm repositories..."
9  helm repo add rancher-latest https://releases.rancher.com/server-charts/latest
10 helm repo add jetstack https://charts.jetstack.io
11
12 # Update Helm repositories
13 echo "Updating Helm repositories..."
14 helm repo update
15
16 # Add the cert-manager CRD
17 echo "Applying cert-manager CRD..."
18
19 # Install or upgrade cert-manager
20 echo "Installing or upgrading cert-manager..."
21 helm upgrade -i cert-manager jetstack/cert-manager --namespace cert-manager --create-namespace --set crds.
22
```

Hình 4.27: Khai báo các biến, thêm và cập nhật repo, cài đặt hoặc nâng cấp cert-manager

- Cài đặt hoặc nâng cấp Rancher: Triển khai Rancher với các cấu hình:
 - Namespace: cattle-system.
 - Hostname: Tên miền Rancher (rancher.mylab.com).
 - Bootstrap Password: admin.

- Replicas: 1 (triển khai 1 bản sao Rancher).
- Service Type: LoadBalancer.

```
# Install or upgrade Rancher
echo "Installing or upgrading Rancher..."
helm upgrade -i rancher rancher-latest/rancher \
--create-namespace \
--namespace cattle-system \
--set hostname="${HOSTNAME}" \
--set bootstrapPassword="${BOOTSTRAP_PASSWORD}" \
--set replicas=1 \
--set service.type="LoadBalancer"
```

Hình 4.28: Cài đặt hoặc nâng cấp Rancher với cấu hình đã chọn

- Hoàn tất cài đặt: In thông báo "Installation completed." sau khi hoàn thành.

4.3.2. Monitoring:

Sử dụng hai công cụ chính Prometheus và Grafana.

Cài đặt Prometheus và Grafana thông qua Helm [5]:

```
1 helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
2 helm repo update
3 kubectl create ns monitoring
4 helm upgrade --install prometheus prometheus-community/kube-prometheus-stack --namespace monitoring
```

Hình 4.29: Lệnh Helm cài đặt Kube-Prometheus-stack

Kết quả khi chạy lệnh helm:

```
root@master1:/home/ubuntu# helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
kubectl create ns monitoring
helm upgrade --install prometheus prometheus-community/kube-prometheus-stack --namespace monitoring
"prometheus-community" has been added to your repositories
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "jetstack" chart repository
...Successfully got an update from the "apache-airflow" chart repository
...Successfully got an update from the "twent" chart repository
...Successfully got an update from the "longhorn" chart repository
...Successfully got an update from the "joxit" chart repository
...Successfully got an update from the "community-charts" chart repository
...Successfully got an update from the "rancher-latest" chart repository
...Successfully got an update from the "jupyter" chart repository
...Successfully got an update from the "jupyterhub" chart repository
...Successfully got an update from the "prometheus-community" chart repository
...Successfully got an update from the "bitnami" chart repository
Update Complete. #Happy Helmng!#
namespace/monitoring created
Release "prometheus" does not exist. Installing it now.
NAME: prometheus
LAST DEPLOYED: Thu Jan  9 12:20:47 2025
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace monitoring get pods -l "release=prometheus"
Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to create & configure Alertmanager and Prometheus instances using the Operator.
```

Hình 4.30: Kết quả khi chạy lệnh helm

4.3.3. Longhorn

Cài đặt Longhorn - một giải pháp lưu trữ phân tán trên Kubernetes, sử dụng Helm với các cấu hình tùy chỉnh.

- Cấu hình biến môi trường:
 - LONGHORN_NAMESPACE: Namespace mặc định (longhorn-system).
 - INGRESS_HOST: Hostname cho Ingress (longhorn.mylab.com).
- Thêm và cập nhật Helm repository: Thêm Helm repo của Longhorn (<https://charts.longhorn.io>) và cập nhật repo.

```
$ 3-install_longhorn.sh ×  
tasks > bash_script > $ 3-install_longhorn.sh  
1  #!/bin/bash  
2  
3  # Set variables for your environment  
4  export LONGHORN_NAMESPACE="longhorn-system"  
5  export INGRESS_HOST="longhorn.mylab.com"  
6  
7  # Add and update the Helm repository  
8  echo "Adding Longhorn Helm repository..."  
9  helm repo add longhorn https://charts.longhorn.io  
10 helm repo update
```

Hình 4.31: Cấu hình biến môi trường, thêm và cập nhật Helm repo

- Tạo namespace Longhorn: Sử dụng kubectl create namespace.
- Cài đặt Longhorn: Triển khai Longhorn với Helm:
 - Kích hoạt Ingress và đặt hostname.
 - Bật xác nhận khi xóa (deletingConfirmationFlag).
 - Replica mặc định: 1.
 - Chính sách thu hồi (Reclaim Policy): Delete.

```

# Create the Longhorn namespace
echo "Creating Longhorn namespace..."
kubectl create namespace $LONGHORN_NAMESPACE

# Install Longhorn using Helm with additional configurations
echo "Installing Longhorn with custom settings..."
helm upgrade -i longhorn longhorn/longhorn --namespace $LONGHORN_NAMESPACE \
    --set ingress.enabled=true \
    --set ingress.host=$INGRESS_HOST \
    --set defaultSettings.deletingConfirmationFlag=true \
    --set persistence.defaultStorageClass.replicaCount=1 \
    --set persistence.defaultClassReplicaCount=1 \
    --set persistence.reclaimPolicy="Delete"

```

Hình 4.32: Tạo namespace và cài đặt Longhorn

- Kiểm tra trạng thái: Chờ 30 giây để triển khai hoàn tất.
- Kiểm tra trạng thái các pods trong namespace longhorn-system.

```

# Wait for the deployment to finish
echo "Waiting for the Longhorn deployment to complete..."
sleep 30

# Verify the status of Longhorn pods
echo "Verifying Longhorn pod status..."
kubectl get pods --namespace $LONGHORN_NAMESPACE

```

Hình 4.33: Kiểm tra trạng thái Longhorn

4.3.4. Metallb

- Triển khai MetalLB (Load Balancer cho Kubernetes) và triển khai ứng dụng mẫu với LoadBalancer Service [6].
- Cài đặt MetalLB: Sử dụng kubectl để áp dụng cấu hình MetalLB từ một tệp YAML trên GitHub.
- Đặt dải địa chỉ IP: Thiết lập biến môi trường cho dải địa chỉ IP từ 192.168.9.111 đến 192.168.9.147.
- Tạo cấu hình MetalLB: Tạo tệp metallb-config.yaml với một IPAddressPool và một L2Advertisement.
- Áp dụng cấu hình MetalLB: Sử dụng kubectl để áp dụng cấu hình mới tạo.

- Kiểm tra trạng thái: Thực hiện lệnh kubectl get all -n metallb-system để xem tất cả các tài nguyên trong namespace metallb-system.
- Tạo ứng dụng mẫu: Tạo tệp demo.yaml định nghĩa một Deployment và một Service kiểu LoadBalancer.
- Triển khai ứng dụng: Áp dụng tệp demo.yaml để triển khai ứng dụng và dịch vụ.

```
$ 4-install_metallb.sh ×
tasks > bash_script > $ 4-install_metallb.sh
1  #!/bin/bash
2
3  kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.14.9/config/manifests/metallb-native
4
5  export DEFAULT_IP_RANGE_START=192.168.9.111
6  export DEFAULT_IP_RANGE_END=192.168.9.147
7
8  cat <<EOF > metallb-config.yaml
9  apiVersion: metallb.io/v1beta1
10 kind: IPAddressPool
11 metadata:
12   name: first-pool
13   namespace: metallb-system
14 spec:
15   addresses:
16     - ${DEFAULT_IP_RANGE_START}-${DEFAULT_IP_RANGE_END}
17   ---
18  apiVersion: metallb.io/v1beta1
19  kind: L2Advertisement
20  metadata:
21    name: example
22    namespace: metallb-system
23 EOF
24
25 kubectl apply -f metallb-config.yaml
26
27 kubectl get all -n metallb-system
```

Hình 4.34: Cài đặt và cấu hình Metallb

4.3.5. Docker Registry + UI

4.3.5.1. Docker Registry

- Tạo PersistentVolumeClaim (PVC): Tạo PVC cho Docker Registry với Longhorn và dung lượng 10Gi.

```

cat <<EOF > pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: docker-registry-pvc
  namespace: mlops
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: longhorn
EOF

```

Hình 4.35: Tạo PVC cho Docker Registry

- Cài đặt Helm Repo và Docker Registry: Cài đặt Helm repo twuni và tìm kiếm chart Docker Registry [7].

```

helm repo add twuni https://helm.twun.io
helm repo update
helm search repo docker-registry

```

Hình 4.36: Thêm và update Helm repo

- Cấu hình Docker Registry qua values.yaml: Tạo file values.yaml để cấu hình Docker Registry với LoadBalancer, PVC, ingress, và các thiết lập bảo mật.
- Cài đặt Docker Registry: Sử dụng Helm để cài đặt Docker Registry với cấu hình từ values.yaml.

```
helm upgrade --install docker-registry twuni/docker-registry -f values.yaml --namespace mlops
```

Hình 4.37: Cài đặt Docker Registry

- Kiểm tra dịch vụ Docker Registry: Kiểm tra dịch vụ Docker Registry sau khi cài đặt.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
docker-registry	LoadBalancer	10.43.35.133	192.168.9.111	5000:30612/TCP	7h45m
docker-registry-ui-docker-registry-ui-user-interface	LoadBalancer	10.43.127.70	192.168.9.115	80:32283/TCP	8s
hub	ClusterIP	10.43.175.251	<none>	8081/TCP	76m
mlflow	LoadBalancer	10.43.134.241	192.168.9.113	5000:30943/TCP	152m
my-release-minio	LoadBalancer	10.43.240.36	192.168.9.112	9000:31697/TCP,9001:30687/TCP	5h49m
postgres-service	ClusterIP	10.43.54.186	<none>	5432/TCP	7h31m
proxy-api	ClusterIP	10.43.161.150	<none>	8001/TCP	76m
proxy-public	LoadBalancer	10.43.136.178	192.168.9.114	80:31355/TCP	76m

Hình 4.38: Kiểm tra dịch vụ Docker Registry

- Đẩy và kéo hình ảnh Docker: Đẩy hình ảnh Docker lên Docker Registry và kéo lại từ registry.

```
root@master1:/home/ubuntu/container_registry# docker pull nginx:latest
Docker tag nginx:latest registry.local/my-nginx:test
Docker push registry.local/my-nginx:test
latest: Pulling from library/nginx
fd674058ff8f: Already exists
66e42bcee1c: Pull complete
b99b9c5d9e5: Pull complete
d98674871f5: Pull complete
e109dd2a0d7: Pull complete
a8cc133ff82: Pull complete
44f27309ea1: Pull complete
Digest: sha256:42e917aaa1b5bb40dd0f6f7f4f857490ac7747d7ef73b391c774a41a8b994f15
Status: Downloaded newer image for nginx:latest
Docker.io/library/nginx:latest
```

Hình 4.39: Tải image Nginx mới nhất

```
root@master1:/home/ubuntu/container_registry# docker tag nginx:latest registry.local:5000/my-nginx:test
root@master1:/home/ubuntu/container_registry# docker push registry.local:5000/my-nginx:test
The push refers to repository [registry.local:5000/my-nginx]
af90855d8344: Pushed
ad206e285c61: Pushed
24aeff94f79e: Pushed
d567f5b4517e: Pushed
14a96b2ac595: Pushed
c4c8312766f1: Pushed
8b296f486960: Pushed
test: digest: sha256:1fdc65e25b1aa5ec3774c6226f5cf2d537c83bf42cf8ed679554489bfda6c385 size: 1778
```

Hình 4.40: Gắn tag cho image Nginx và push lên Registry

```
docker pull registry.local:5000/my-nginx:test
Untagged: nginx:latest
Untagged: nginx@sha256:42e917aaa1b5bb40dd0f6f7f4f857490ac7747d7ef73b391c774a41a8b994f15
Untagged: registry.local:5000/my-nginx:test
Untagged: registry.local:5000/my-nginx@sha256:1fdc65e25b1aa5ec3774c6226f5cf2d537c83bf42cf8ed679554489bfda6c385
test: Pulling from my-nginx
Digest: sha256:1fdc65e25b1aa5ec3774c6226f5cf2d537c83bf42cf8ed679554489bfda6c385
Status: Downloaded newer image for registry.local:5000/my-nginx:test
registry.local:5000/my-nginx:test
```

Hình 4.41: Pull Image Nginx về máy

- Cấu hình Docker và RKE2 để sử dụng Registry:
 - Cập nhật hoặc tạo file cấu hình daemon.json để thêm registry vào danh sách các registry không an toàn.
 - Cập nhật hoặc tạo file cấu hình registries.yaml để thêm registry vào RKE2

```
cat <<EOF > /etc/docker/daemon.json
{
  "insecure-registries": ["registry.mylab.com:5000"]
}
EOF
```

Hình 4.42: Cấu hình daemon.json

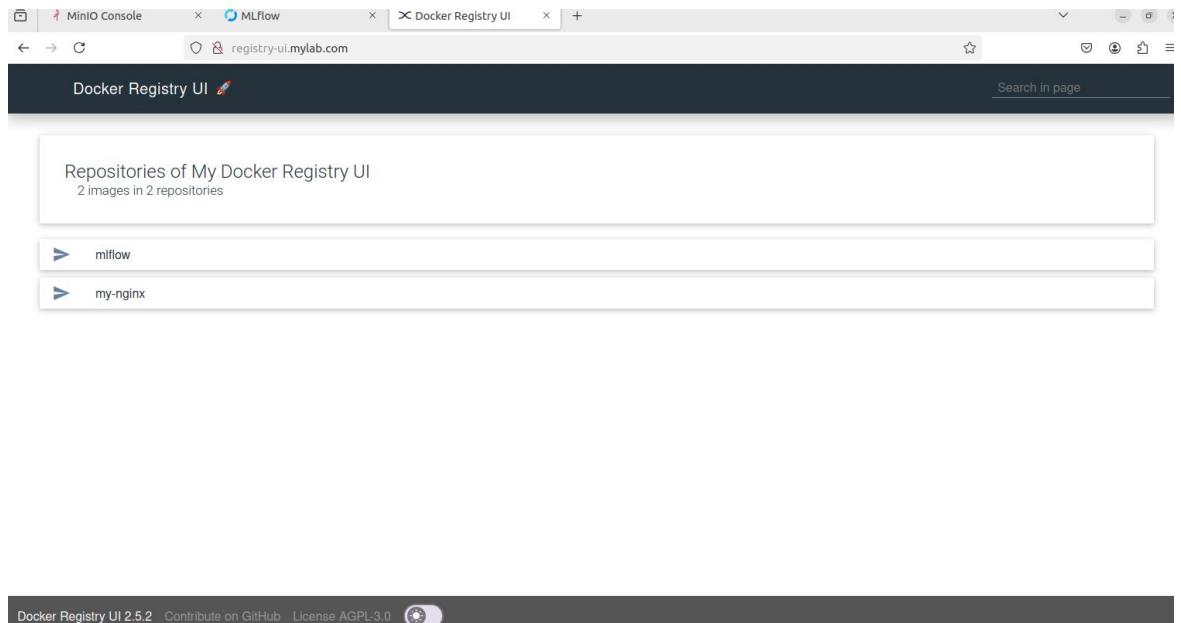
```
cat <<EOF > /etc/rancher/rke2/registries.yaml
mirrors:
  - registry.mylab.com:5000:
    endpoint:
      - "http://registry.mylab.com:5000"
EOF
```

Hình 4.43: Cấu hình registries.yaml trong RKE2

- Khởi động lại Docker và RKE2: Restart dịch vụ Docker và RKE2 để áp dụng các thay đổi.

4.3.5.2. Docker Registry UI

- Thêm repo và cập nhật.
- Tạo ConfigMap cho Docker Registry UI.
- Cấu hình giá trị cho Docker Registry UI.
- Cài đặt Docker Registry UI qua Helm.



Hình 4.44: Giao diện của Docker Registry UI

4.4. Triển khai các ứng dụng học máy trên Cluster

4.4.1. PostgreSQL

- Tạo Namespace: Tạo namespace mllops để chứa các tài nguyên.
- Tạo PersistentVolumeClaim: Tạo một PersistentVolumeClaim tên là postgres-pvc với dung lượng 5Gi, sử dụng storageClassName: longhorn.

```

# Create namespace
kubectl create namespace mlops

# Create PersistentVolumeClaim
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
  namespace: mlops
spec:
  storageClassName: longhorn
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
EOF

```

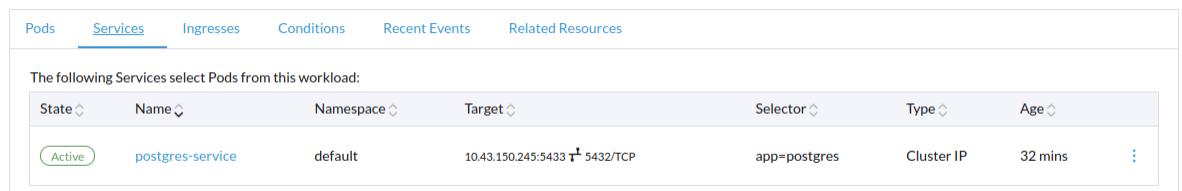
Hình 4.45: Tạo namespace, PVC

- Tạo Deployment: Triển khai một Deployment tên là postgres-deployment trong namespace mlops, chạy một container PostgreSQL phiên bản 14.



Hình 4.46: Kiểm tra Deployment trong Rancher

- Tạo Service: Tạo một Service kiểu ClusterIP tên là postgres-service để truy cập vào PostgreSQL trên cổng 5432.



Hình 4.47: Kiểm tra Service trong Rancher

- Cài đặt PostgreSQL Client: Cập nhật và cài đặt PostgreSQL client (bước này có thể được kích hoạt nếu chạy trên máy cục bộ).

- Hướng dẫn kết nối: In ra hướng dẫn để kết nối đến PostgreSQL, bao gồm cách tìm địa chỉ IP bên ngoài của service và các lệnh SQL mẫu để tạo bảng và chèn dữ liệu.

4.4.2. MinIO

- Thêm Helm Repo Bitnami: Cập nhật repository Helm của Bitnami helm.twun.io[8].
- Tạo PersistentVolumeClaim (PVC): Tạo PVC cho MinIO với Longhorn và dung lượng 10Gi.

```
$ 2-install_minio.sh M
tasks > bash_script > helm-install-mlops > $ 2-install_minio.sh
1   helm repo add bitnami https://charts.bitnami.com/bitnami
2
3   helm repo update
4
5   cat <<EOF | kubectl apply -f -
6   apiVersion: v1
7   kind: PersistentVolumeClaim
8   metadata:
9     name: minio-pvc
10    namespace: mlops
11   spec:
12     storageClassName: longhorn
13     accessModes:
14       - ReadWriteOnce
15     resources:
16       requests:
17         storage: 10Gi
18 EOF
```

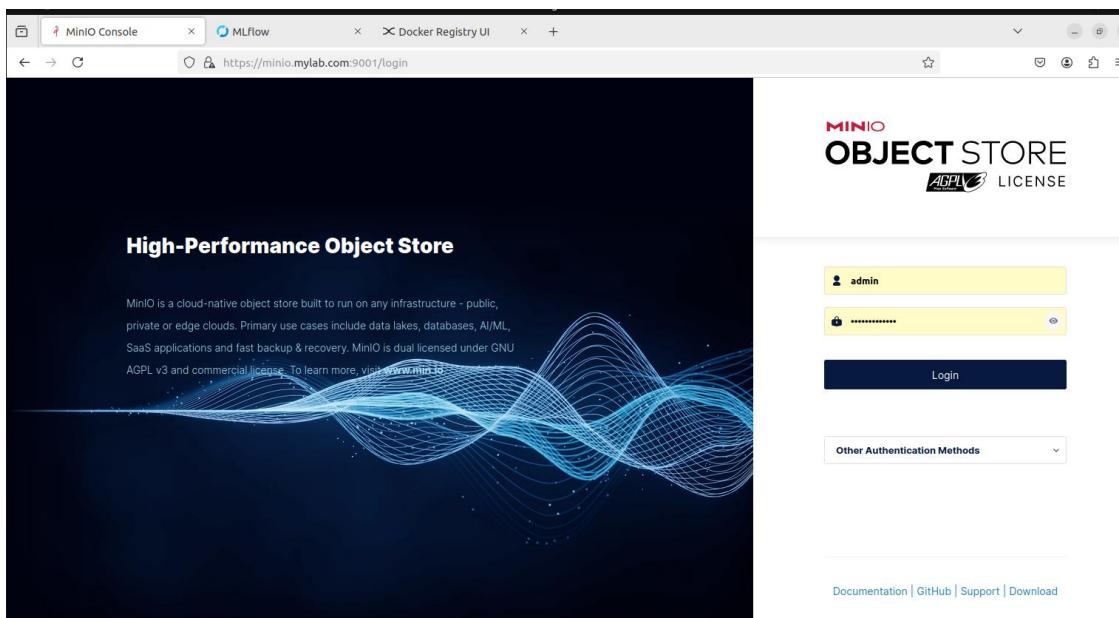
Hình 4.48: Thêm Helm repo, tạo PVC

- Tạo Secret cho MinIO: Mã hóa và tạo secret chứa tên người dùng và mật khẩu root của MinIO.

```
echo -n 'admin' | base64  
echo -n 'adminpassword' | base64  
  
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Secret  
metadata:  
  name: minio-root-user  
  namespace: mlops  
data:  
  root-user: YWRtaW4=  
  root-password: YWRtaW5wYXNzd29yZA==  
EOF
```

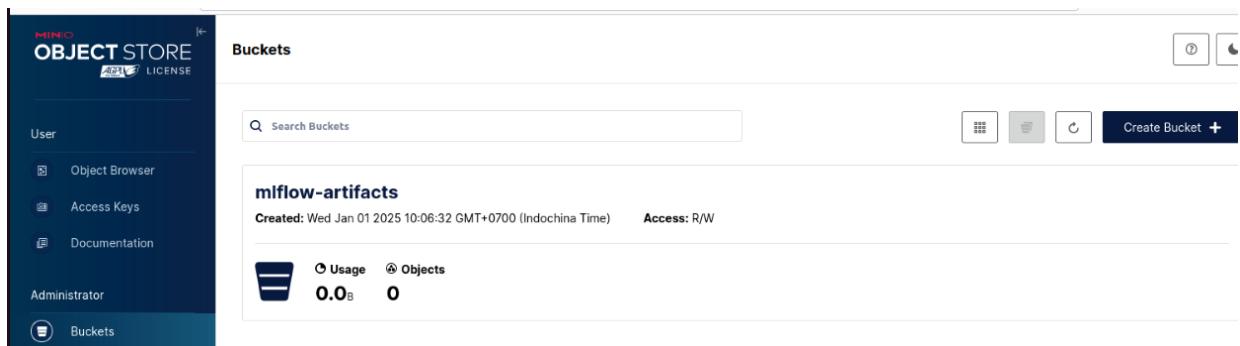
Hình 4.49: Tạo Secret

- Cấu hình MinIO qua values.yaml: Tạo file values.yaml cấu hình cho MinIO, bao gồm cài đặt lưu trữ, dịch vụ, ingress, bảo mật, và tài nguyên.
- Cài đặt MinIO qua Helm: Cài đặt MinIO sử dụng Helm với cấu hình từ values.yaml.
- Lấy thông tin người dùng và mật khẩu MinIO: Giải mã thông tin người dùng và mật khẩu từ secret.
- Chạy MinIO Client: Sử dụng MinIO Client để kết nối đến MinIO server và thiết lập alias.
- Truy cập MinIO UI: Địa chỉ truy cập MinIO UI qua Ingress (minio.mylab.com:9001).



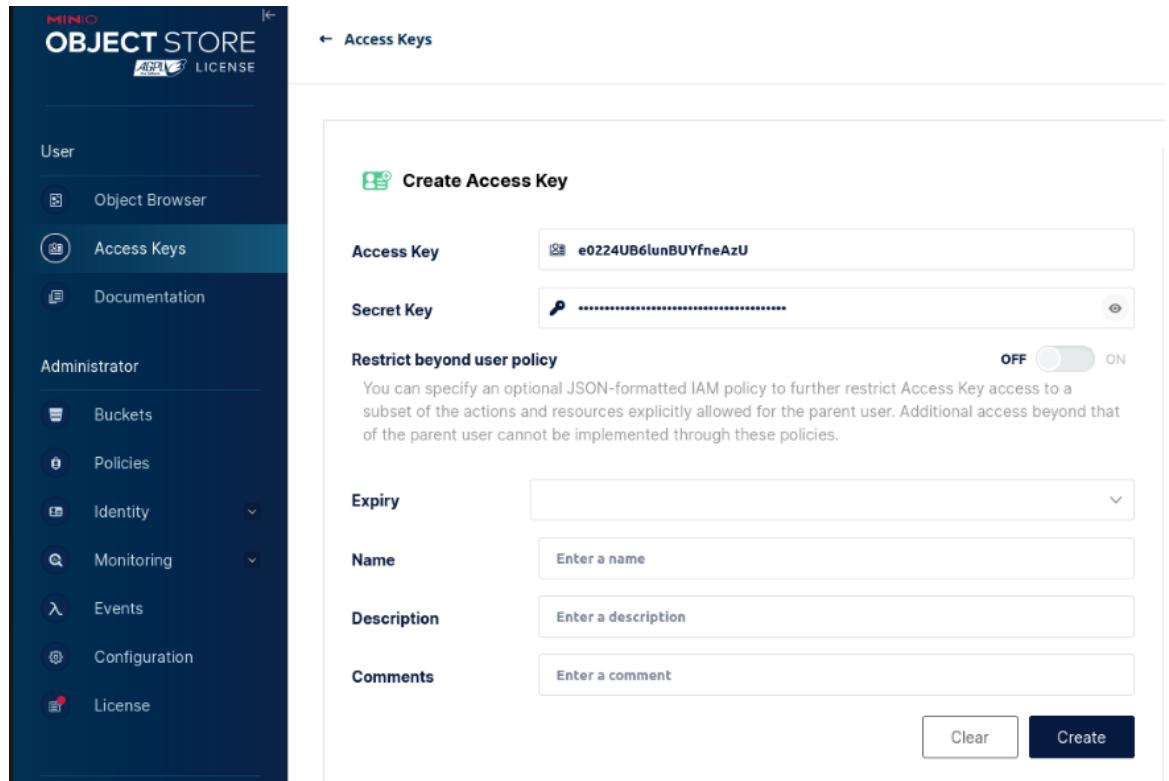
Hình 4.50: Trang đăng nhập của MinIO

- Tạo Bucket với tên là “mlflow-artifacts”.



Hình 4.51: Tạo Bucket

- Tạo Access Key và copy lại Access Key và Secret Key



Hình 4.52: Tạo Access Key

4.4.3. MLflow

- Thêm repo và cập nhật [9].
- Tạo cơ sở dữ liệu PostgreSQL cho MLflow

```
helm repo add community-charts https://community-charts.github.io/helm-charts

helm repo update

cat <<EOM
psql -h 10.43.178.165 -p 5432 -U postgres
Password1234
CREATE DATABASE mlflow_db;
CREATE USER mlflow_user WITH PASSWORD 'mlflow_password';
GRANT ALL PRIVILEGES ON DATABASE mlflow_db TO mlflow_user;

CREATE DATABASE mlflow_auth_db;
CREATE USER auth_user WITH PASSWORD 'auth_password';
GRANT ALL PRIVILEGES ON DATABASE mlflow_auth_db TO auth_user;

\1
EOM
```

Hình 4.53: Tạo Database trong PostgreSQL cho MLflow

- Dockerfile cho MLflow

- Xây dựng và đẩy Docker image cho MLflow

```
cat <<EOF > Dockerfile
FROM ghcr.io/mlflow/mlflow:v2.19.0

# Install system dependencies for PostgreSQL and S3 support, as well as Python dependencies
RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
    python3-dev \
    build-essential \
    pkg-config && \
    rm -rf /var/lib/apt/lists/* && \
    # Upgrade pip and install Python dependencies
    pip install --upgrade pip && \
    pip install psycopg2-binary boto3

# Default command to start bash (or you can modify it to run MLflow server if needed)
CMD ["bash"]

EOF

# Build and push the Docker image to your private registry
docker build -t registry.mylab.com:5000/mlflow:v2 -f Dockerfile .
docker push registry.mylab.com:5000/mlflow:v2
```

Hình 4.54: Tạo và đẩy image của MLflow lên Docker Registry

Tạo Persistent Volume Claim (PVC) cho MLflow

```
cat <<EOF > pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mlflow-pvc
  namespace: mlops
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: longhorn
EOF

kubectl apply -f pvc.yaml
```

Hình 4.55: Tạo PVC cho MLflow

- Cấu hình Helm cho MLflow

```

cat <<EOF > values.yaml
replicaCount: 1

# Image of mlflow
image:
  repository: registry.mylab.com:5000/mlflow
  # -- The docker image pull policy
  pullPolicy: Always
  # -- The docker image tag to use. Default app version
  tag: "v2"

```

Hình 4.56: Cấu hình để sử dụng image MLflow trên Docker Registry

```

backendStore:
  databaseConnectionCheck: true
  postgres:
    enabled: true
    host: "10.43.178.165"
    port: 5432
    database: "mlflow_db"
    user: "mlflow_user"
    password: "mlflow_password"
    driver: "psycopg2"

artifactRoot:
  s3:
    enabled: true
    bucket: "mlflow-artifacts"
    awsAccessKeyId: "CeGeT5fesfdHa4unYt2p"
    awsSecretAccessKey: "ZxxHWqpOfy8SGzkzGbjMkG86tudAq9KAiPckB5gJ"
extraEnvVars:
  MLFLOW_S3_ENDPOINT_URL: "https://minio.mylab.com:9000"
  MLFLOW_S3_IGNORE_TLS: true

```

Hình 4.57: Cấu hình PostgreSQL và s3 với tham số tương ứng trong hệ thống

- Cài đặt MLflow với Helm

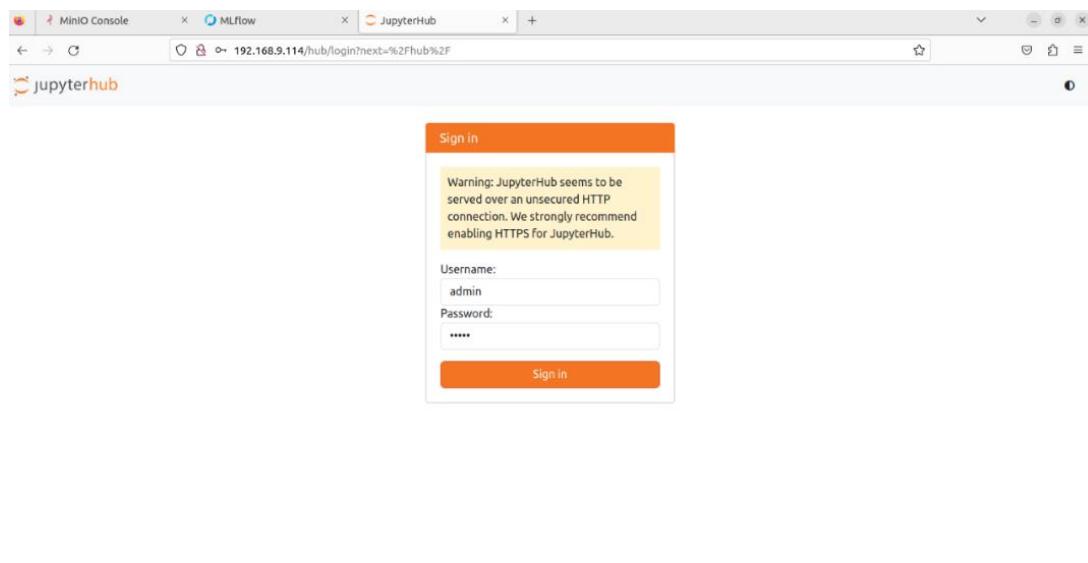
4.4.4. JupyterHub

- Thêm repo và cập nhật.
- Lấy giá trị mặc định của chart JupyterHub.
- Cài đặt JupyterHub với Helm.

```
$ 5-install_jupyterhub.sh X
tasks > bash_script > helm-install-mlops > $ 5-install_jupyterhub.sh
3 helm repo add jupyterhub https://jupyterhub.github.io/helm-chart/
4 helm repo update
5
6 helm show values jupyterhub/jupyterhub > values.yaml
7
8 helm upgrade --cleanup-on-fail --install my-jupyter jupyterhub/jupyterhub \
9   --namespace mlops \
10  --values values.yaml
```

Hình 4.58: Cài đặt JupyterHub với Helm

Hình 4.59: Kết quả cài đặt JupyterHub với Helm



Hình 4.60: Trang đăng nhập JupyterHub

Chương 5. Thực nghiệm và đánh giá

5.1. Kịch bản thực nghiệm

5.1.1. Triển khai và kiểm thử tính năng HA

- Mục tiêu: Kiểm tra khả năng chịu lỗi và tính sẵn sàng cao (HA) của hệ thống.
- Cách thực hiện:
 - Triển khai cluster RKE2 với 3 master nodes và 2 worker nodes.
 - Cấu hình Keepalived cho master nodes để đảm bảo tính sẵn sàng khi có sự cố xảy ra.
 - Mô phỏng sự cố bằng cách tắt một trong các master nodes và kiểm tra khả năng phục hồi của hệ thống.

5.1.2. Quản lý tài nguyên Cluster với Rancher

- Mục tiêu: Kiểm tra khả năng quản lý tài nguyên (CPU, RAM) và khả năng điều chỉnh tài nguyên cho các ứng dụng trong Kubernetes bằng Rancher.
- Cách thực hiện:
 - Triển khai Rancher để quản lý cluster Kubernetes và thiết lập các cấu hình tài nguyên cho pod.
 - Giới hạn CPU và RAM cho các pod sử dụng các tham số requests và limits trong Kubernetes.
 - Tạo một số tác vụ và theo dõi việc Rancher có thể quản lý tài nguyên trong cluster như thế nào khi có sự thay đổi tải công việc.

The screenshot shows the Rancher Workloads interface. At the top, there are buttons for Redeploy, Download YAML, and Delete. To the right is a 'Create' button and a 'Filter' input field. Below is a table with the following data:

State	Name	Namespace	Type	Image	Restarts	Age	Health
Active	minio-deployment	mlops	Deployment	quay.io/minio/minio	0	49 mins	<div style="width: 100%;"><div style="width: 100%;"> </div></div>
Active	postgres-deployment	mlops	Deployment	postgres:13	0	59 mins	<div style="width: 100%;"><div style="width: 100%;"> </div></div>
Active	registry-deployment	mlops	Deployment	registry:2.8.3	0	58 secs	<div style="width: 100%;"><div style="width: 100%;"> </div></div>

Hình 5.1: Theo dõi trạng thái Deployment của các công cụ

5.1.3. Quản lý lưu trữ với Longhorn

- Mục tiêu: Kiểm tra khả năng quản lý và lưu trữ dữ liệu phân tán với Longhorn trong môi trường Kubernetes.
- Cách thực hiện:
 - Cài đặt và cấu hình Longhorn để cung cấp giải pháp lưu trữ phân tán cho các pod trong Kubernetes.
 - Triển khai các pod yêu cầu lưu trữ để kiểm tra khả năng truy xuất và lưu trữ dữ liệu từ Longhorn.
 - Mô phỏng việc tăng khối lượng dữ liệu và kiểm tra khả năng mở rộng của Longhorn.

5.1.4. Triển khai và quản lý các tác vụ học máy bằng MLflow, MinIO và PostgreSQL

- Mục tiêu: Đánh giá khả năng triển khai các ứng dụng học máy sử dụng các công cụ MLflow, MinIO và PostgreSQL trong Kubernetes.
- Cách thực hiện:
 - Cài đặt và cấu hình MinIO để cung cấp dịch vụ lưu trữ đối tượng cho các dữ liệu học máy.
 - Triển khai PostgreSQL để lưu trữ các dữ liệu có cấu trúc và hỗ trợ việc quản lý thông tin liên quan đến các mô hình học máy.
 - Triển khai MLflow để theo dõi, quản lý và lưu trữ các mô hình học máy trong quá trình huấn luyện.
 - Kiểm tra khả năng giao tiếp giữa các công cụ này và theo dõi hiệu suất trong suốt quá trình huấn luyện mô hình học máy.

```

root@master1:/home/ubuntu# psql -h 127.0.0.1 -p 5433 -U postgres -W
Password:
psql (14.15 (Ubuntu 14.15-0ubuntu0.22.04.1), server 13.18 (Debian 13.18-1.pgdg120+1))
Type "help" for help.

postgres=# CREATE TABLE test_table(id SERIAL PRIMARY KEY, name VARCHAR(50));
INSERT INTO test_table(name) VALUES ('Kubernetes Persistent Volume');
INSERT INTO test_table(name) VALUES ('Kubernetes Persistent Volume Claim');
SELECT * FROM test_table;
CREATE TABLE
INSERT 0 1
INSERT 0 1
 id |          name
----+-----
  1 | Kubernetes Persistent Volume
  2 | Kubernetes Persistent Volume Claim
(2 rows)

postgres=#

```

Hình 5.2: Kết nối đến PostgreSQL service và tạo Table

```

python3 -m venv env
source env/bin/activate

pip3 install --upgrade pip
pip3 install mlflow
pip3 install boto3
pip3 install scikit-learn

export MLFLOW_TRACKING_URI=http://mlflow.mylab.com:5000
export MLFLOW_TRACKING_INSECURE_TLS=true
export MLFLOW_S3_ENDPOINT_URL=https://minio.mylab.com:9000
export MLFLOW_S3_IGNORE_TLS=true
export MLFLOW_ARTIFACTS_DESTINATION=s3://mlflow-artifacts
export AWS_ACCESS_KEY_ID=CeGeT5fesfdHa4unYt2p
export AWS_SECRET_ACCESS_KEY=ZxxHwqpOfy8SGzkzGbjMkG86tudAq9KAiPckB5gJ

git clone https://github.com/mlflow/mlflow
python3 mlflow/examples/sklearn_elasticnet_wine/train.py

```

Hình 5.3: Câu lệnh để chạy ví dụ với MLflow

```

(env) root@master1:/home/ubuntu/mlflow-testing# export MLFLOW_TRACKING_URI=http://mlflow.mylab.com:5000
export MLFLOW_TRACKING_INSECURE_TLS=true
export MLFLOW_S3_ENDPOINT_URL=https://192.168.9.112:9000
export MLFLOW_S3_IGNORE_TLS=true
export MLFLOW_ARTIFACTS_DESTINATION=s3://mlflow-artifacts
export AWS_ACCESS_KEY_ID=CeGeT5fesfdHa4unYt2p
export AWS_SECRET_ACCESS_KEY=ZxxHwqpOfy8SGzkzGbjMkG86tudAq9KAiPckB5gJ

git clone https://github.com/mlflow/mlflow
python3 mlflow/examples/sklearn_elasticnet_wine/train.py
fatal: destination path 'mlflow' already exists and is not an empty directory.
Elasticnet model (alpha=0.500000, l1_ratio=0.500000):
  RMSE: 0.7931640229276851
  MAE: 0.6271946374319586
  R2: 0.10862644997792614
Successfully registered model 'ElasticnetWineModel'.
2025/01/03 13:11:50 INFO mlflow.store.model_registry.abstract_store: Waiting up to 300 seconds for model version to finish creation. Model name: ElasticnetWineModel, version 1
Created version '1' of model 'ElasticnetWineModel'.
👉 View run smilling-seal-154 at: http://mlflow.mylab.com:5000/#/experiments/0/runs/acie49e65a31462490197b6adfc1b2cb
(env) root@master1:/home/ubuntu/mlflow/mlflow-testing# 

```

Hình 5.4: Kết quả của ví dụ trên CLI

Hình 5.5: Kết quả của Experiment trên MLflow UI

Hình 5.6: Object được cập nhật sau các lần sử dụng MLflow trên MinIO UI

Hình 5.7: Chi tiết về các Artifacts

The screenshot shows the MLflow interface at mlflow.mylab.com:5000/#/experiments/1/runs/cf14f754b1ad4e07a1bc1. The experiment name is "default-params". The "Overview" tab is selected. The "Description" section contains a note: "No description". The "Details" section provides the following information:

Created at	2025-01-03 13:52:23
Created by	root
Experiment ID	1
Status	Finished
Run ID	cf14f754b1ad4e07a1bc16a63d0f6f94
Duration	5.1s
Datasets used	dataset (2276d121) Train +1
Tags	<code>estimator_name: ElasticNet</code> <code>estimator_class: sklearn.linear_model.coorda...</code>
Source	b.py
Logged models	sklearn
Registered models	—

Hình 5.8: Thông tin Experiment

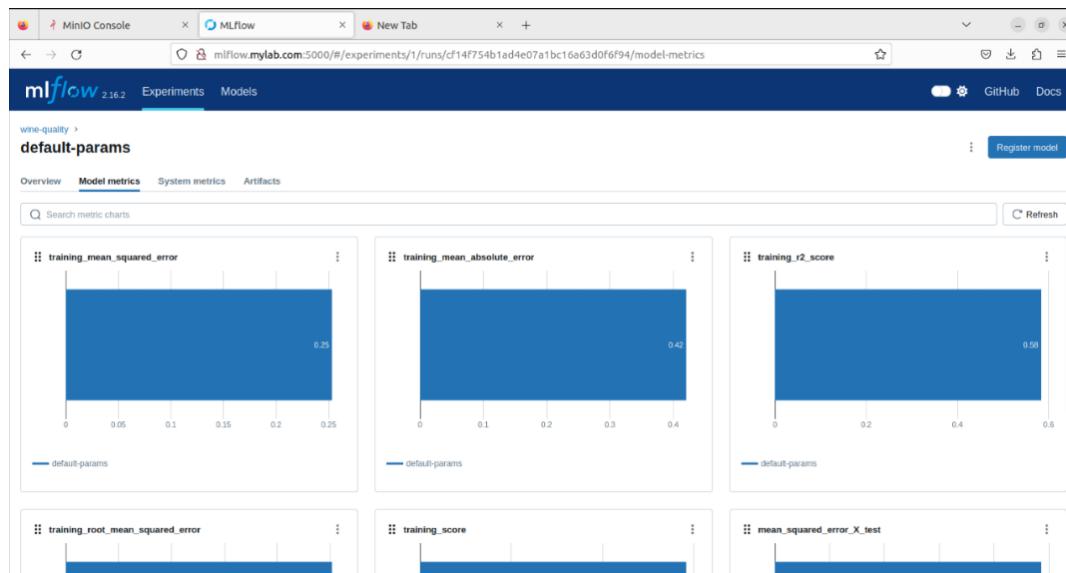
The screenshot shows the MLflow interface at mlflow.mylab.com:5000/#/experiments/1/runs/cf14f754b1ad4e07a1bc16a63d0f6f94. The experiment name is "default-params". The "Overview" tab is selected. The "Parameters (11)" section lists:

Parameter	Value
alpha	1.0
copy_X	True
fit_intercept	True
l1_ratio	0.5
max_iter	1000
positive	False
precompute	False
random_state	None
selection	cyclic
tol	0.0001
warm_start	False

The "Metrics (8)" section lists:

Metric	Value
training_mean_squared_error	0.25307968551506227
training_mean_absolute_error	0.42040967653556716
training_r2_score	0.5834828287052534
training_root_mean_squared_error	0.5030702590245842
training_score	0.5834828287052534
mean_squared_error_X_test	0.24291669149166806
mean_absolute_error_X_test	0.423323955405606
r2_score_X_test	0.5662201937648783

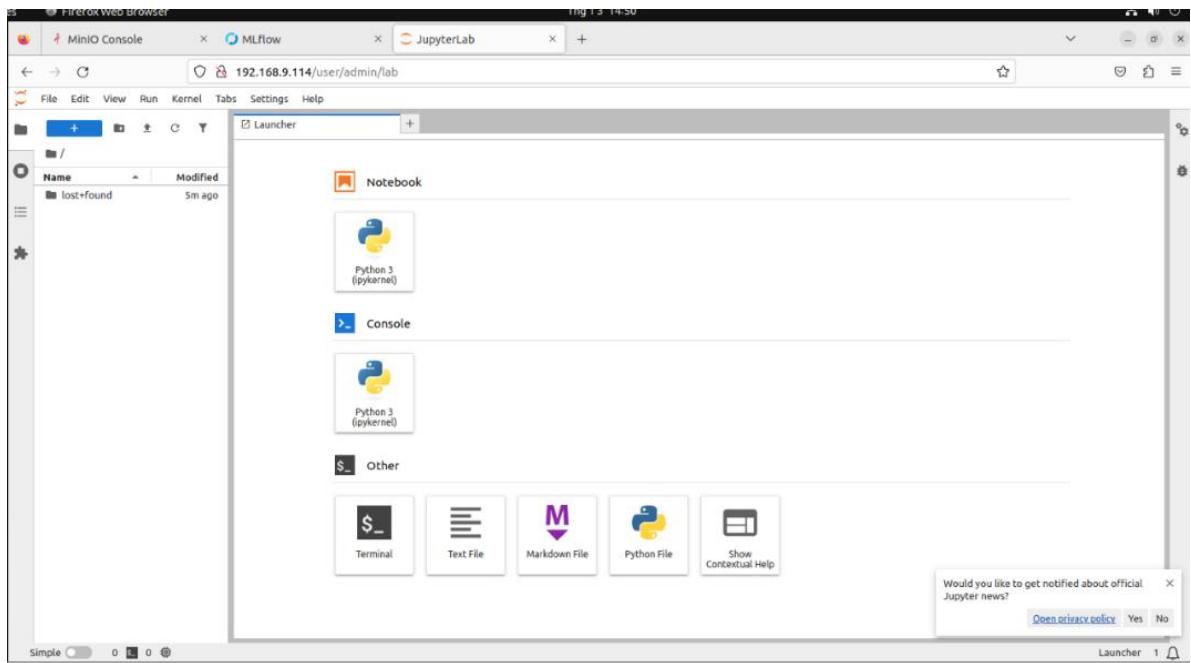
Hình 5.9: Thông Tin Chi Tiết Experiment



Hình 5.10: Biểu đồ hiển thị hiệu suất của các thí nghiệm hoặc các tham số

5.1.5. Triển khai học máy bằng JupyterHub

- Mục tiêu: Kiểm tra khả năng triển khai môi trường học máy tương tác bằng JupyterHub trong Kubernetes.
- Cách thực hiện:
 - Cài đặt và cấu hình JupyterHub để cung cấp môi trường học máy tương tác cho người dùng.
 - Triển khai các notebook Jupyter để chạy các tác vụ học máy và thử nghiệm mô hình.
 - Theo dõi hiệu suất sử dụng tài nguyên và quản lý các notebook trong quá trình huấn luyện mô hình.

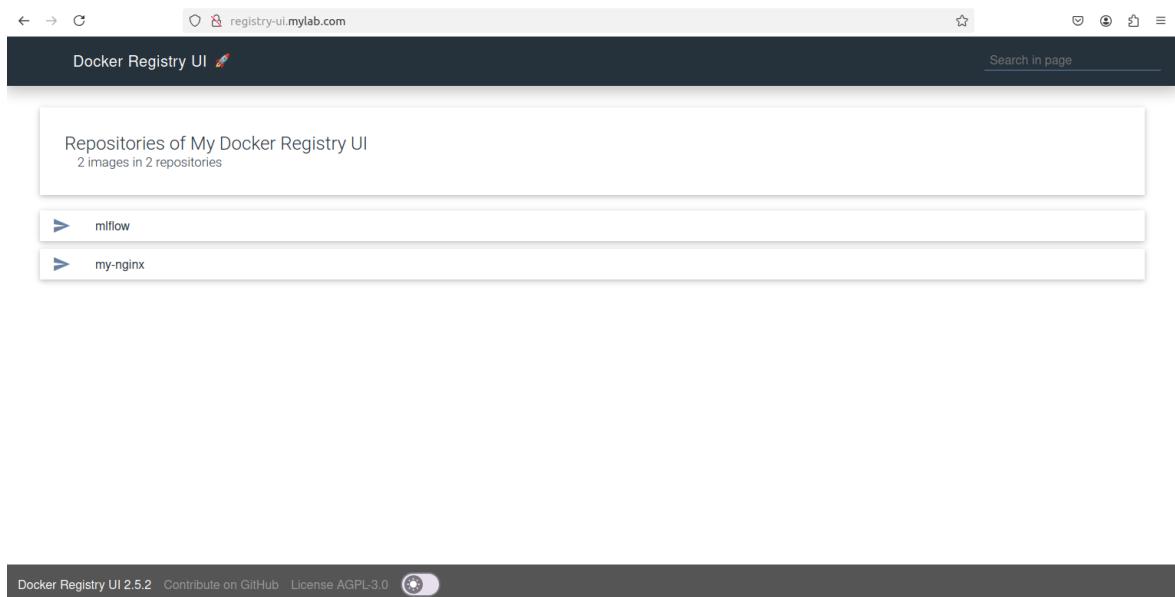


Hình 5.11: Trang chủ JupyterHub

5.1.6. Giám sát và cảnh báo sử dụng Kube-Prometheus Stack

- Mục tiêu: Kiểm tra khả năng giám sát và gửi cảnh báo trong Kubernetes sử dụng Kube-Prometheus Stack.
- Cách thực hiện:
 - Cài đặt và cấu hình Kube-Prometheus Stack bao gồm Prometheus, Grafana và Alertmanager để giám sát tài nguyên và hoạt động của các pod.
 - Thiết lập các cảnh báo để phát hiện các vấn đề như tải CPU cao, sử dụng RAM vượt quá giới hạn, hoặc sự cố trong các pod.
 - Mô phỏng các tình huống lỗi như quá tải tài nguyên và kiểm tra xem các cảnh báo có được gửi kịp thời hay không.

5.1.7. Triển khai Docker Registry và UI



Hình 5.12: Docker Registry UI cập nhật các image được push

Mục tiêu: Đánh giá khả năng triển khai Docker Registry và UI để quản lý và lưu trữ các hình ảnh Docker trong Kubernetes.

- Cách thực hiện:
 - Cài đặt Docker Registry trong Kubernetes để lưu trữ các hình ảnh Docker của các ứng dụng, bao gồm cả các mô hình học máy và các phần mềm hỗ trợ.
 - Cài đặt Docker Registry UI để dễ dàng quản lý và xem các hình ảnh Docker đã được lưu trữ.
 - Kiểm tra việc đẩy và kéo hình ảnh Docker vào Docker Registry từ các node trong Kubernetes.
 - Mô phỏng các tình huống như việc xóa hình ảnh, tải lại hình ảnh và kiểm tra độ tin cậy của Docker Registry trong việc duy trì hình ảnh Docker.

5.1.8. Triển khai và kiểm thử MetalLB cho Load Balancing

```
root@master1:/home/ubuntu# kubectl get svc -n default
NAME         TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP  10.43.0.1    <none>        443/TCP   20h
loadbalancer  LoadBalancer 10.43.212.215  192.168.9.111  80:31152/TCP  16m
```

Hình 5.13: Test thử tính năng cung cấp dịch vụ LoadBalancer của Metallb

- Mục tiêu: Đánh giá khả năng cung cấp dịch vụ cân bằng tải (load balancing) cho các ứng dụng trong Kubernetes sử dụng MetalLB.
- Cách thực hiện:
 - Cài đặt và cấu hình MetalLB trong Kubernetes để cung cấp dịch vụ Load Balancer cho các ứng dụng.
 - Triển khai một số ứng dụng yêu cầu truy cập qua Load Balancer (ví dụ: web server hoặc ứng dụng học máy).
 - Kiểm tra tính năng tự động phân phối tải của MetalLB bằng cách gửi các yêu cầu đồng thời đến các ứng dụng và theo dõi cách thức phân phối tải.
 - Mô phỏng tình huống lỗi bằng cách tắt một trong các node và kiểm tra khả năng phân phối lại tải của MetalLB.

5.2. Kết quả thực nghiệm

5.2.1. Đánh giá kết quả thực nghiệm

Các kết quả thực nghiệm được đánh giá dựa trên các tiêu chí sau:

- MetallB sẽ phân phối tải một cách hiệu quả giữa các pod trong Kubernetes và đảm bảo khả năng chịu tải cao cho các ứng dụng.
- Docker Registry và UI sẽ hoạt động ổn định, cho phép người dùng dễ dàng quản lý và truy xuất các hình ảnh Docker trong môi trường Kubernetes.
- Prometheus + Grafana triển khai thành công trên hệ thống và hoạt động ổn định. Các cảnh báo sẽ được gửi kịp thời khi có sự cố xảy ra và hiển thị trong Grafana một cách rõ ràng.
- JupyterHub sẽ cung cấp môi trường học máy tương tác hiệu quả, hỗ trợ chạy các tác vụ học máy mà không gặp vấn đề về tài nguyên.
- Hệ thống sẽ có khả năng lưu trữ và quản lý dữ liệu học máy cũng như các mô hình trong một môi trường phân tán và dễ mở rộng nhờ vào PostgreSQL + MinIO + MLflow.

- Longhorn sẽ cung cấp khả năng lưu trữ phân tán hiệu quả và có thể mở rộng khi khối lượng dữ liệu tăng.
- Các pod không vượt quá giới hạn tài nguyên đã được định nghĩa trong Rancher, và Rancher sẽ quản lý tài nguyên hiệu quả.
- Hệ thống cần tiếp tục hoạt động bình thường mà không bị gián đoạn dịch vụ nhờ vào cấu hình HA.

5.2.2. So sánh với các công cụ/công trình liên quan

- RKE2 vs Kubernetes distributions khác (K3s, Minikube): RKE2 vượt trội trong việc triển khai HA, tích hợp dễ dàng với các công cụ quản lý và lưu trữ, với khả năng chịu lỗi và tính sẵn sàng cao hơn so với K3s và Minikube, vốn phù hợp cho môi trường nhỏ hoặc thử nghiệm.
- Rancher vs Kubernetes Dashboard: Rancher cung cấp giao diện người dùng dễ sử dụng và nhiều tính năng quản lý, giám sát toàn diện hơn Kubernetes Dashboard, giúp quản lý tài nguyên và tích hợp công cụ mạnh mẽ như Longhorn, MetalLB.
- Có HA vs Không có HA: Hệ thống có HA (với RKE2, Keepalived) cung cấp tính sẵn sàng cao, khả năng mở rộng và độ tin cậy vượt trội, đảm bảo không gián đoạn dịch vụ khi gặp sự cố. Hệ thống không có HA dễ gặp sự cố và gián đoạn dịch vụ khi một master node gặp vấn đề.

Chương 6. Kết luận và hướng phát triển

6.1. Kết luận

Đồ án "Điều phối container và quản lý tài nguyên cho các tác vụ học máy với Kubernetes" đã thực hiện và đánh giá một hệ thống container orchestration mạnh mẽ với Kubernetes, sử dụng các công cụ hỗ trợ như RKE2, Rancher, Longhorn, MetalLB, Postgres, MinIO, MLflow, JupyterHub, và Kube-Prometheus stack để phục vụ các tác vụ học máy. Hệ thống được triển khai với kiến trúc High Availability (HA) cho các master node sử dụng Keepalived, giúp đảm bảo tính ổn định và khả năng phục hồi cao trong trường hợp các node gặp sự cố.

Các phương pháp nghiên cứu và công cụ triển khai đã được thực hiện một cách chi tiết qua các kịch bản thực nghiệm, bao gồm:

Triển khai HA cho Kubernetes Cluster: Kiểm thử khả năng phục hồi và tính ổn định của hệ thống khi có sự cố xảy ra trên các node master.

- Quản lý tài nguyên (CPU, RAM, Storage): Sử dụng Rancher để giới hạn tài nguyên CPU và RAM cho các pod và sử dụng Longhorn để quản lý storage, đảm bảo các tài nguyên được phân bổ hợp lý cho các tác vụ học máy.
- Triển khai học máy: Triển khai các ứng dụng học máy với MLflow, MinIO, và PostgreSQL, giúp quản lý mô hình và dữ liệu học máy một cách hiệu quả.
- Giám sát và cảnh báo: Sử dụng Kube-Prometheus stack để giám sát và cảnh báo về tình trạng của các ứng dụng trong Kubernetes.

Mặc dù đã triển khai thành công các công cụ và tính năng chính của hệ thống, một số công cụ bổ sung như ArgoCD, NeuVector, Istio, Airflow, và KServe vẫn chưa được tích hợp hoàn toàn vào cluster. Tuy nhiên, các script cài đặt cho những công cụ này đã được chuẩn bị và có thể dễ dàng tích hợp trong tương lai.

6.2. Hướng phát triển

Hệ thống đã triển khai là nền tảng vững chắc để mở rộng và phát triển thêm trong tương lai. Các hướng phát triển tiềm năng bao gồm:

- Tích hợp ArgoCD, NeuVector, Istio, Airflow, và KServe: Hoàn thiện việc tích hợp các công cụ này vào cluster để tối ưu hóa quy trình phát triển và triển khai ứng dụng, đặc biệt là trong việc tự động hóa CI/CD (Continuous Integration/Continuous Deployment) với ArgoCD và bảo mật hệ thống với NeuVector.
- Cải thiện khả năng mở rộng (Scalability): Mở rộng hệ thống Kubernetes bằng cách bổ sung thêm các node worker và tối ưu hóa cấu hình tài nguyên, giúp hệ thống đáp ứng nhu cầu ngày càng tăng từ các tác vụ học máy.
- Tăng cường giám sát và cảnh báo: Phát triển các kịch bản giám sát nâng cao với Prometheus và Grafana để theo dõi chi tiết hơn về hiệu suất và tình trạng của các ứng dụng học máy, đồng thời cải thiện các cảnh báo để đưa ra phản ứng kịp thời với các vấn đề.
- Ứng dụng Deep Learning và AI: Tích hợp thêm các công cụ và framework hỗ trợ các tác vụ deep learning (như TensorFlow, PyTorch, hoặc Keras) để mở rộng khả năng hỗ trợ các mô hình học máy phức tạp hơn.
- Tăng cường bảo mật và phân quyền: Nâng cao các tính năng bảo mật trong hệ thống bằng cách sử dụng Istio cho quản lý mạng dịch vụ và triển khai các chính sách bảo mật chi tiết hơn cho các container và Kubernetes clusters.

TÀI LIỆU THAM KHẢO

- [1] *Kubernetes Official Documentation.* <https://kubernetes.io/docs/home/>
- [2] Introduction (2025) RKE2. <https://docs.rke2.io/>
- [3] Ansible Role for RKE2. <https://github.com/lablabs/ansible-role-rke2>
- [4] Rancher Helm Chart on Artifact Hub.
<https://artifacthub.io/packages/helm/rancher-latest/rancher>.
- [5] *Kube-Prometheus-Stack Helm Chart on Artifact Hub.*
<https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack>
- [6] *MetalLB Official Documentation.* <https://metallb.io>
- [7] *Docker Registry UI Helm Chart on Artifact Hub.*
<https://artifacthub.io/packages/helm/joxit/docker-registry-ui>
- [8] *MinIO Helm Chart on Artifact Hub.*
<https://artifacthub.io/packages/helm/bitnami/minio>
- [9] *MLflow Helm Chart Values File.* <https://github.com/community-charts/helm-charts/blob/main/charts/mlflow/values.yaml>